Consistent Feature-Model Driven Software Product Line Evolution

Von der

Carl-Friedrich-Gauß-Fakultät

der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines

Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von

Michael Nieke

geboren am 10.06.1989

in Northeim

Eingereicht am:	08.01.2021
Disputation am:	18.12.2020
1. Referentin:	Prof. Dr.-Ing. Ina Schaefer
2. Referent:	Prof. Dr. Bernhard Rumpe

2021

Abstract

Software Product Lines (SPLs) are an approach to manage families of closely related software systems in terms of configurable functionality. A *feature model* captures common and variable functionalities of an SPL on a conceptual level in terms of *features*. Reusable artifacts, such as code, documentation, or tests are related to features using a *feature-artifact mapping*. A *product* or *variant* of an SPL can be derived by selecting features in a *configuration* which is used in combination with the feature-artifact mapping to collect a set of reusable artifacts. With an automatic generation mechanism, the set of reusable artifacts are composed to a product.

Over the course of time, SPLs and their artifacts are subject to change. As SPLs are particularly complex, their evolution is a challenging task. Consequently, SPL evolution must be thoroughly planned well in advance. However, plans typically do not turn out as expected and, thus, replanning is required. Feature models lean themselves for driving SPL evolution as they serve as a main communication artifact and represent functionality on an abstract level without technical details. However, replanning of feature-model evolution can lead to inconsistencies as the basis for already planned subsequent evolution steps changes. Moreover, feature-model *anomalies* that are an indicator for errors may be introduced during evolution. Current feature modeling techniques are not able to ensure feature-model consistency in presence of replanning, and feature-model anomalies that have been introduced during evolution cannot be fixed efficiently.

Along with feature-model evolution, other SPL artifacts need to consistently evolve; especially for configurations as changes to an SPL may result in different behavior of a product resulting from an existing configuration. As different engineer roles that are responsible for performing SPL evolution (domain engineers) and maintaining configurations (application engineers) typically cannot communicate with each other, product behavior may even change unnoticed. As a result, products with changed behavior may be deployed which may lead to failures and, consequently, to additional costs.

The work of this thesis provides remedy to the aforementioned challenges by presenting an approach for consistent evolution of SPLs. The main contributions of this thesis can be distinguished into three key areas: planning and replanning feature-model evolution, analyzing feature-model evolution, and consistent SPL artifact evolution. As a starting point for SPL evolution, we introduce *Temporal Feature Models (TFMs)* that allow capturing the entire evolution timeline of a feature model in one artifact, i.e., past history, present changes, and planned evolution steps. Furthermore, temporal relations between the changes are modeled as first-class entity such that replanning and introduction of intermediate evolution steps is possible.

We provide an execution semantics of feature-model evolution operations that guarantees consistency of feature-model evolution timelines. To this end, we formalize feature-model evolution timelines and feature-model evolution operations in terms of Structural Operational Semantics (SOS). To keep feature models free from anomalies, we introduce analyses to detect

anomalies in feature-model evolution timelines. Additionally, we devise a concept for explaining anomalies in terms of causing evolution operations.

To enable consistent SPL artifact evolution, we generalize the concept of modeling evolution timelines in TFMs to be applicable for any modeling language. Such a uniform language concept to capture evolution forms the basis to keep all SPL artifacts consistent, i.e., in a compatible state. Moreover, we provide a methodology that enables domain engineers to guide application engineers through configuration evolution by sharing knowledge on SPL evolution and by defining automatically applicable configuration update operations with the goal of preserving product behavior.

With the presented concepts, SPL evolution can be planned in terms of consistent and anomaly-free TFMs. Consistent evolution of other SPL artifacts can be captured with a particular focus on knowledge transfer from domain engineers to application engineers for updating configurations. All concepts of this thesis are implemented in the tool suite DarwinSPL using a model-driven engineering approach. The concepts and their implementation are evaluated using publicly available case studies.

Zusammenfassung

Mit Hilfe von Softwareproduktlinien (SPLs) kann konfigurierbare Funktionalität von eng verwandten Softwaresystemen verwaltet werden. In einem *Feature Modell* werden gemeinsame und variable Funktionalitäten einer SPL auf Basis abstrakter *Features* modelliert. Wiederverwendbare Artefakte, wie Code, Dokumentation oder Tests, werden in einem *Feature-Artefakt Mapping* Features zugeordnet. Ein *Produkt* oder eine *Variante* einer SPL kann abgeleitet werden, indem Features in einer Konfiguration ausgewählt werden, was gemeinsam mit dem *Feature-Artefakt Mapping* genutzt wird, um eine Menge von Artefakten abzuleiten. Diese Artefaktmenge wird daraufhin in einem automatischen Generierungsprozess genutzt, um das finale Produkt zu erzeugen.

Im Laufe der Zeit müssen sich SPLs und deren Artefakte verändern. Da SPLs ganze Softwarefamilien modellieren, ist deren Evolution eine besonders herausfordernde Aufgabe, die gründlich im Voraus geplant werden muss. Jedoch können Pläne meist nicht wie ursprünglich geplant umgesetzt werden, und die Evolution muss umgeplant werden. Feature Modelle eignen sich besonders als Planungsmittel einer SPL, da sie ein zentrales Kommunikationsartefakt aller Beteiligten sind und Funktionalität auf einem abstrakten Level ohne technische Details darstellen. Umplanung von Feature Modell Evolution kann jedoch zu Inkonsistenzen führen, da sich die Basis für bereits geplante zukünftige Änderungsschritte ändert. Außerdem können Feature Modell *Anomalien* im Zuge der Evolution eingeführt werden, welche Indikatoren für Fehler sind. Existierende Techniken zur Feature Modellierung betrachten weder die Umplanung von Evolution, noch die Sicherstellung der Konsistenz von Feature Modellen. Auch können Anomalien, die durch Evolution eingeführt wurden, nicht effizient repariert werden.

Im Anschluss an die Feature Modell Evolution muss die Evolution anderer SPL Artefakte konsistent modelliert werden. Konfigurationen sind besonders von SPL Evolution betroffen, da die resultierenden Produkte ein anderes Verhalten als vor der Evolution aufweisen können. Domänen-Ingenieure, welche die Evolution einer SPL durchführen, und Applikations-Ingenieure, welche Konfigurationen verwalten, können typischerweise nicht miteinander kommunizieren. Dies kann zu unbemerkten Verhaltensänderungen von Produkten existierender Konfigurationen führen, die in Fehlern und entsprechenden zusätzlichen Kosten resultieren.

In dieser Arbeit wird ein Ansatz zur konsistenten Evolution von SPLs vorgestellt, der die zuvor genannten Herausforderungen adressiert. Die Beiträge dieser Arbeit lassen sich in drei Kernbereiche aufteilen: Planung und Umplanung von Feature Modell Evolution, Analyse von Feature Modell Evolution und konsistente Evolution von SPL Artefakten. *Temporal Feature Models (TFMs)* werden als Startpunkt für SPL Evolution eingeführt. In einem TFM wird die gesamte Evolutionszeitlinie eines Feature Modells in einem Artefakt abgebildet, was sowohl vergangene Änderungen, den aktuellen Zustand, als auch geplante Änderungen beinhaltet. Darüber hinaus werden zeitliche Relationen von Änderungen als eigenständige Entität modelliert, was die Umplanung und Einführung von Zwischenevolutionsschritten ermöglicht.

Auf Basis einer Ausführungssemantik wird die Konsistenz von Feature Modell Evolutionszeitlinien sichergestellt. Dazu werden Feature Modell Evolutionszeitlinien und zugehörige Evolutionsoperationen mit Hilfe von Structural Operational Semantics (SOS) formalisiert. Um Feature Modelle frei von Anomalien zu halten, werden Analysen eingeführt, welche die gesamte Evolutionszeitlinie eines Feature Modells auf Anomalien untersucht. Um gefundene Anomalien zu erklären, werden die verursachenden Evolutionsoperationen identifiziert.

Das Konzept zur Modellierung von Feature Modell Evolutionszeitlinien aus TFMs wird verallgemeinert, um die gesamte Evolution von Modellen beliebiger Modellierungssprachen spezifizieren zu können. Ein entsprechendes einheitliches Konzept zur Modellierung von Evolution stellt die Basis dar, um alle SPL Artefakte konsistent, das heißt in einem kompatiblen Zustand, zu halten. Des Weiteren wird eine Methodik vorgestellt, die es Domänen-Ingenieuren ermöglicht, Applikations-Ingenieure bei der Evolution von Konfigurationen anzuleiten. Dies geschieht, indem Wissen über die SPL Evolution geteilt wird und automatisch anwendbare Aktualisierungsoperationen für Konfigurationen definiert werden mit dem Ziel, das Verhalten von Produkten stabil zu halten.

Die Konzepte, die in dieser Arbeit vorgestellt werden, erlauben es, SPL Evolution mit Hilfe von konsistenten und anomalielosen TFMs zu planen. Darüber hinaus können wir die anderer SPL Artefakte modellieren mit Fokus auf dem Wissenstransfer von Domänen- zu Applikations-Ingenieuren. Alle Konzepte wurden im Werkzeug DarwinSPL auf Basis eines modellgetriebenen Ansatzes entwickelt. Mit Hilfe von frei verfügbaren Fallstudien wurden die Konzepte und deren Implementierung evaluiert.

Acknowledgements

In the years of working on my thesis, many important people have supported me and I would like to dedicate the following lines to them.

First of all, I want to thank Ina Schaefer for convincing me to proceed with studying after finishing my Bachelor's and to start with my Ph.D. after my Master's. I also thank her for giving me the chance to do my Ph.D. at her institute and for providing much freedom. I thank my reviewer Prof. Bernhard Rumpe for taking the effort of reviewing my dissertation.

With my colleagues at the Institute of Software Engineering and Automotive Informatics, going to work was a real pleasure and always fun. I would like to thank them for the great atmosphere at the institute and for the academic feedback. I especially thank Christoph Seidl for guiding me through this tough time as a mentor, giving me directions for my research, and being a good friend. Without Christoph, I would not have come this far. I would also like to thank my former student assistant and current colleague Adrian Hoff for his support, including implementing parts of DarwinSPL, beautiful graphics, and videos for our successful YouTube channel.

I thank my loving parents for fostering my education and for their ongoing help to learn life. My mother Barbara was care-taking in each section of my life and always had an open ear. My feather Reinhard taught me critical and thorough thinking. Together they laid the foundation for my career. I also would like to thank my brother Sebastian for enduring me as a teenager and for being a good friend.

Finally, I thank my beloved wife Melanie for being my anchor and supporting me throughout the years. I thank Melanie and my daughter Miriam for reminding me what are the most important things in life, and for being the sunshine in my life.

Publications

This doctoral thesis is based on the following peer-reviewed publications ordered by relevance, starting with the most relevant.

[1] **M. Nieke**, J. Mauro, C. Seidl, T. Thüm, I. C. Yu, and F. Franzke. "Anomaly Analyses for Feature-model Evolution". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: ACM, 2018, pp. 188–201. DOI: 10.1145/3278122.3278123.

[2] **M. Nieke**, G. Sampaio, T. Thüm, C. Seidl, L. Teixeira, and I. Schaefer. "GuyDance: Guiding the Evolution of Product-Line Configurations". In: *Proceedings of the 24th International Systems and Software Product Line Conference - Volume B*. SPLC '20. Accepted for publication. 2020. DOI: https://doi.org/10.1145/3382026.3425769.

[3] **M. Nieke**, G. Sampaio, T. Thüm, C. Seidl, L. Teixeira, and I. Schaefer. "Guided Configuration Evolution for Product Lines". In: *Software and Systems Modeling* (2020). Submitted.

[4] A. Hoff, **M. Nieke**, C. Seidl, E. H. Saether, I. M. Sandberg, C. C. Din, I. C. Yu, and I. Schaefer. "Consistency-Preserving Evolution Planning on Feature Models". In: *Proceedings of the 24th International Systems and Software Product Line Conference - Volume A*. SPLC '20. Montréal, Canada: Association for Computing Machinery, 2020. DOI: https://doi.org/10.1145/3382025.3414964.

[5] **M. Nieke**, C. Seidl, and T. Thüm. "Back to the Future: Avoiding Paradoxes in Feature-model Evolution". In: *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 2*. SPLC '18. Gothenburg, Sweden: ACM, 2018, pp. 48–51. DOI: 10.1145/3236405.3237201.

[6] **M. Nieke**, A. Hoff, C. Seidl, and I. Schaefer. "Automated Metamodel Augmentation for Seamless Model Evolution Tracking and Planning". In: *Journal of Computer Languages* (2020). In minor revision.

[7] **M. Nieke**, A. Hoff, and C. Seidl. "Automated Metamodel Augmentation for Seamless Model Evolution Tracking and Planning". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 68–80. DOI: 10.1145/3357765.3359526.

[8] **M. Nieke**, G. Engel, and C. Seidl. "DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-aware Software Product Lines". In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '17. Eindhoven, Netherlands: ACM, 2017, pp. 92–99. DOI: 10.1145/3023956.3023962.

[9] **M. Nieke**, C. Seidl, and S. Schuster. "Guaranteeing Configuration Validity in Evolving Software Product Lines". In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 73–80. DOI: 10.1145/2866614.2866625.

[10] D. Hinterreiter, **M. Nieke**, L. Linsbauer, C. Seidl, H. Prähofer, and P. Grünbacher. "Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 115–128. DOI: 10.1145/3357765.3359515.

Further peer-reviewed publications directly related to this thesis.

[11] J. Mauro, **M. Nieke**, C. Seidl, and I. C. Yu. "Anomaly Detection and Explanation in Context-Aware Software Product Lines". In: *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*. SPLC '17. Sevilla, Spain: ACM, 2017, pp. 18–21. DOI: 10.1145/3109729.3109752.

[12] J. Mauro, **M. Nieke**, C. Seidl, and I. C. Yu. "Context-aware reconfiguration in evolving software product lines". In: *Science of Computer Programming* 163 (2018), pp. 139–159. DOI: https://doi.org/10.1016/j.scico.2018.05.002.

[13] J. Sprey, C. Sundermann, S. Krieter, **M. Nieke**, J. Mauro, T. Thüm, and I. Schaefer. "SMT-Based Variability Analyses in FeatureIDE". In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS '20. Magdeburg, Germany: Association for Computing Machinery, 2020. DOI: 10.1145/3377024.3377036.

Further peer-reviewed publications indirectly related to this thesis.

[14] J. Mauro, **M. Nieke**, C. Seidl, and I. C. Yu. "Context Aware Reconfiguration in Software Product Lines". In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '16. Salvador, Brazil: Association for Computing Machinery, 2016, pp. 41–48. DOI: 10.1145/2866614.2866620.

[15] **M. Nieke**, J. Mauro, C. Seidl, and I. C. Yu. "User Profiles for Context-Aware Reconfiguration in Software Product Lines". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*. Ed. by T. Margaria and B. Steffen. Cham: Springer International Publishing, 2016, pp. 563–578. DOI: 10.1007/978-3-319-47169-3_44.

[16] C. Pietsch, C. Seidl, **M. Nieke**, and T. Kehrer. "Chapter 8 - Delta-oriented development of model-based software product lines with DeltaEcore and SiPL: A comparison". In: *Model Management and Analytics for Large Scale Systems*. Ed. by B. Tekinerdogan, Ö. Babur, L. Cleophas, M. van den Brand, and M. Akşit. Academic Press, 2020, pp. 167–201. DOI: https://doi.org/10.1016/B978-0-12-816649-9.00017-X.

[17] C. Chesta, F. Damiani, L. Dobriakova, M. Guernieri, S. Martini, **M. Nieke**, V. Rodrigues, and S. Schuster. "A Toolchain for Delta-Oriented Modeling of Software Product Lines". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by T. Margaria and B. Steffen. Cham: Springer International Publishing, 2016, pp. 497–511. DOI: 10.1007/978-3-319-47169-3_40.

[18] B. Caesar, **M. Nieke**, A. Köcher, C. Hildebrandt, C. Seidl, A. Fay, and I. Schaefer. "Context-sensitive reconfiguration of collaborative manufacturing systems". In: *IFAC-PapersOnLine* 52.13 (2019). 9th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2019, pp. 307–312. DOI: https://doi.org/10.1016/j.ifacol.2019.11.194.

[19] S. Schuster, **M. Nieke**, and I. Schaefer. "Name Resolution Strategies in Variability Realization Languages for Software Product Lines". In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. FOSD 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 11–17. DOI: 10.1145/3001867.3001869.

[20] S. Holthusen, **M. Nieke**, T. Thüm, and I. Schaefer. "Proof-Carrying Apps: Contract-Based Deployment-Time Verification". In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*. Ed. by T. Margaria and B. Steffen. Cham: Springer International Publishing, 2016, pp. 839–855. DOI: 10.1007/978-3-319-47166-2_58.

Contents

List of Abbreviations

CTL Computation Tree Logic

EMF Eclipse Modeling Framework

MOF Meta Object Facility

OCL Object Constraint Language

OMG Object Management Group

SFT Software Fault Tree

SOS Structural Operational Semantics

SPL Software Product Line

TFM Temporal Feature Model

VCS Version Control System

Part I.

Context and Preliminaries

1 Introduction

A *Software Product Line (SPL)* is an approach for managed large-scale reuse for software families [PBL05, SRC+12]. An SPL facilitates engineers to model commonalities and variabilities, allowing to generate *variants* of a family of software systems using reusable artifacts. For instance, the Linux kernel is used in many systems, such as car infotainment systems, televisions, smartphones, personal computers, or Internet routers. As the Linux kernel is an SPL, commonalities can be reused for the different systems, and variable aspects can be captured explicitly. After configuring variable aspects, a software variant for a concrete kernel can be generated. A variability model, such as a *feature model*, is a technology to represent variability of an SPL on a conceptual level in terms of configurable features and constraints between them [PBL05, SRC+12, KCH+90, ABK+16]. A feature diagram represents a feature model visually in a tree-like notation. For instance, for the Linux kernel, many variants require to communicate using an IP protocol feature and a feature to define firewall rules using the *iptables* functionality. A *configuration* is a set of selected features of a feature model and is *valid* if it satisfies all feature-model constraints [SHT+07]. For instance, a configuration of the Linux kernel selecting the iptables feature can be valid only if it also selects the IP feature. In realization artifacts, such as code or models, variability is implemented using different *variability realization mechanisms*, such as preprocessors [LAL+10], plug-ins [CC06] or delta modeling [SBB+10, CHS11]. In a *feature-artifact mapping*, engineers associate (partial) configurations with (parts of) realization artifacts using Boolean formulas. Using a configuration, the associated realization artifacts can be identified via the feature-artifact mapping and used to generate products [SRC+12, ABK+16, CEC00].

Software evolution and maintenance are some of the most costly and time-consuming tasks when developing software and, thus, they are key disciplines in software engineering [Boe84]. SPLs are especially long-living software systems [BCM+04, FK05] and, thus, many evolution iterations are performed to accommodate new or changed requirements [SB99]. Additionally, SPL evolution is more complex than the evolution of individual software systems as many products can be derived with different requirements and many involved stakeholders [BCM+04, WGS+14, BP14]. Thus, not only bugs are fixed, but also variability changes as new variable options are added, existing options are removed or constraints change. When performing SPL evolution, the impact of the evolution on all existing products and the introduction of new products has to be considered [Liv11, BP14]. Furthermore, an SPL is typically embedded in other business processes, e.g., hardware system development, production processes, or marketing. Consequently, SPL evolution is generally not performed in an ad-hoc manner, but must be thoroughly planned, and interdependent processes need to be adapted subsequently [BPD+10, EBL+10, WGS+14]. In the following, we point out several challenges that arise due to SPL evolution and its planning.

1.1. Motivation

A feature model describes the conceptual part of an SPL and, thus, is the canonical source of variable functionality of an SPL and is used as the main communication artifact. Consequently, SPL evo-

lution optimally starts with changing its feature model which is then propagated to all other parts of the SPL [PCA+13]. Performing feature-model evolution without keeping track of the changes to older versions results in loss of information as the old feature-model version cannot be retrieved. Preserving old feature-model versions can be necessary to support legacy systems that are based on an old SPL version and to provide updates such as bug fixes. As SPLs usually capture large families of software systems, feature-model evolution is a particular complex endeavor and many other processes in a company are affected by the feature-model evolution. Thus, the evolution should be planned thoroughly in advance as opposed to ad-hoc activities to set mid- and long-term evolution goals. Over the course of time, planned evolution then becomes present and, afterwards, past history. Typically, no formal documentation of past changes and planned evolution of feature models exists that describes how evolution has been or should be performed. This information may be crucial to evolve other artifacts or to adapt processes, and to update products to new SPL versions. In the following, we use the term *evolution timeline* to refer to the combination of *planned future evolution*, *present evolution*, and *past evolution history*.

Multiple approaches in research retroactively derive changes between feature-model versions [BKL+16] or automatically track evolution similar to Version Control Systems (VCSs) [ME08, SW16]. These approaches lack the possibility of planning and require additional computation to capture feature-model evolution. Other approaches are able to model feature-model evolution proactively. For instance, Seidl et al. [SSA14b, SSA14c] explicitly model product-line evolution with Hyper Feature Models (HFMs) that introduce feature versions. Each feature version represents a different implementation of a feature. However, with this approach, a feature model's structure cannot be changed. Hinterreiter et al. [HPL+18] introduced the feature-modeling notation FORCE that is capable of capturing feature-model evolution similar to version control systems (i.e., as snapshots) and, consequently, differences between snapshots must be explicitly computed. With EvoFMs, Botterweck et al. [BPP+09, BPD+10, BP14], Schubanz et al. [SPB+12, SPP+13], and Pleuss et al. [PBD+12] introduced a method to capture the evolution of feature models and put a focus on evolution planning. In EvoFMs, the evolution of entire sub-feature models, called feature-model fragments can be modeled using an additional evolution plan. The evolution plan states at which point in time a fragment exists and which evolution operations are applied. EvoFMs have several shortcomings. For one, the evolution of all elements covered by fragments can only be performed together. Thus, if single elements of the fragment should evolve differently than others, the fragment has to be split up. This requires adaptation of the entire EvoFM and the evolution plan. Moreover, operations on the feature-model level, such as move feature, are modeled explicitly. Consequently, exactly these operations must be supported. This also makes analyses complicated as all operations modeled for points in time before the version to analyze must be applied. Finally, it is very complex to have three different models: the original feature model, the EvoFM, and the evolution plan. This makes it very hard for engineers to understand evolution.

Challenge 1: Modeling Feature-Model Evolution Timelines

A modeling notation to track, execute, and plan feature-model evolution as continuous timeline that describes how a feature model changes over time is needed.

SPL evolution planning is an incremental approach that requires continuous adaptation. Even if long-term goals are set by planning feature-model evolution in advance, change requests at short notice affecting parts of the planned evolution may arise; for instance, if the introduction of a feature should be postponed due to a change in a company's strategy. This requires engineers to *replan* evolution steps or to introduce new intermediate evolution steps. Such a changed evolution plan influences other evolution steps planned for later points in time. This may result in inconsistencies as already planned evolution steps for later points in time base on the originally devised change. For instance, if it is planned to add a feature at a certain time point, but if in replanning, its parent feature is deleted, the new feature would not have a parent. Such inconsistencies lead to unusable feature models and, as a consequence, no valid configuration can be generated. Detecting and fixing such inconsistencies is a complex and challenging task as engineers need to know how evolution steps are related to each other and to understand which parts of the feature model were changed in which way by other engineers. Currently, no approach exists that enables to replan feature-model evolution while preventing the introduction of inconsistencies.

General-purpose constraint languages have been developed that enable the definition of model consistency rules. Examples of such languages are the Object Constraint Language (OCL)[1] and Xpand Check language[2]. However, as the languages are very generic, it is not possible to define constraints for particularities such as feature-model evolution replanning. Other research addresses model consistency in general after evolution [GRE10, KKT13]. However, they do not consider the replanning of evolution and, thus, are not able to detect respective inconsistencies. Other research explicitly addresses consistency of SPL artifacts [CP06, VGH+12]. However, these approaches consider neither evolution nor replanning. Guo et al. [GWT+12] explicitly consider consistency in presence of feature-model evolution. However, they do not consider evolution planning and verify only the consistency of a feature model that directly results from applying a feature-model evolution operation. Consequently, further planned evolution steps for subsequent time points are not considered.

Challenge 2: Consistent Feature-Model Evolution Replanning

Methods to prevent the introduction of inconsistencies when replanning feature-model evolution are required.

Proactively revealing mismodeling in the entire feature-model evolution timeline is key to avoid errors in an evolving SPL. Especially in presence of evolution planning, planned changes might base on mismodeled parts that may result in errors at a later point in time. Fixing such errors if they are detected late requires to revert many evolution steps which is expensive and error-prone. A type of mismodeling of feature models are *feature-model anomalies* that are an active field of research [BSR10, GBT+, KAT16, Hem08a, ML04, Bat05, EPH09, FBG+13, KSR13, LSW15, RJW+09, RGM+14, TBR+06, Tri12]. In contrast to the previously mentioned inconsistencies, anomalies are no defects, and in most cases, valid configurations can still be created. However, anomalies (very much like code smells) are possible sources for errors in an SPL. For instance, a *dead feature* anomaly is a non-selectable feature and, thus, variable options are unintentionally lost. If not fixed, many

[1] Object Management Group OMG. Object Constraint Language, Version 2.4 formal/2014-02-01, February 2014
[2] http://www.eclipse.org/modeling/m2t/?project=xpand

anomalies can arise which are even correlated to each other, and it becomes harder to fix them the longer they exist. Thus, keeping feature models anomaly-free is well advised. To fix feature-model anomalies, engineers need to first detect these anomalies and second to understand why these anomalies exist. This is manually infeasible as real-world feature models contain thousands of constraints [KTM+17]. Current methods are able to detect and explain feature-model anomalies, but do not incorporate evolution [BSR10, GBT+, KAT16, Bat05, EPH09, FBG+13, KSR13, LSW15, RJW+09, RGM+14, TBR+06, Tri12]. As feature-model evolution yields another dimension of complexity, it is more likely that engineers inadvertently introduce anomalies during evolution. Hence, detecting and fixing anomalies not just retroactively, after they caused harm, but proactively during the evolution planning of feature models is very useful.

To fix anomalies, engineers use explanations highlighting parts of the feature model that are involved in an anomaly. However, for real-world feature models, explanations can become very large. For instance, for a large feature model from industry (712 features and 1141 constraints), an anomaly explanation involved 92 features and 91 constraints.[3] Consequently, engineers have to inspect all features and constraints to fix the anomaly which results in high effort. However, typically only a few changes of a feature model's evolution result in an anomaly. For instance, the cause for the above-mentioned anomaly in the industry feature model was a recent bug in the tool FeatureIDE that resulted in three changes to feature relations. However, existing approaches to explain anomalies do not incorporate evolution operations [BSR10, GBT+, KAT16, Bat05, EPH09, FBG+13, KSR13, LSW15, RJW+09, RGM+14, TBR+06, Tri12]. For engineers fixing an anomaly, it is crucial to identify evolution operations that cause that anomaly. An anomaly explanation consisting of the causing evolution operations results in a significant explanation length reduction compared to state-of-the-art explanations. Moreover, it is easier for engineers to understand the evolution operations they performed instead of understanding the constraints of a normal anomaly explanation. Such a reduction and simplification is crucial for efficient anomaly handling in the process of evolution.

Analyses that incorporate evolution information typically do not address feature-model anomalies. With the approaches by Alves et al. [AGM+06] and Neves et al. [NBA+15, NTS+11], only changes that do not remove valid configurations from the feature model but potentially add new ones are allowed. This strongly limits potential evolution and is not applicable to real-world evolution scenarios. Other work considers only anomalies that result in no valid configuration of a feature model and, thus, they do not detect other anomalies, such as non-selectable features [SPP+13, SPB+12, PBD+12, BPD+10, GWT+12, GW10]. However, they either do not make use of the information about evolution [SPP+13, SPB+12, PBD+12, BPD+10] or they analyze whether an evolution operation introduces such an anomaly [GWT+12, GW10]. The latter approach requires that the feature model is valid before checking it again and, consequently, they find only the first anomaly. Moreover, none of the mentioned approaches provides anomaly explanations.

Challenge 3: Detecting and Explaining Anomalies in Feature-Model Timelines

Analyses to detect feature-model anomalies in the entire evolution timeline and to explain anomalies in terms of causing evolution operations are needed.

[3] https://github.com/FeatureIDE/FeatureIDE/issues/662

Preserving a consistent SPL state after evolution can be achieved only if all SPL artifacts change in unison. As feature-model evolution should serve as starting point for SPL evolution, other SPL artifacts have to evolve in-line with the feature model. This comprises implementation artifacts, feature-artifact mappings, and configurations. The current practice of using VCSs to capture artifact evolution does not support evolution planning and can emulate it only via workarounds, e.g., by maintaining an additional branch with a planned state. When changing the current version or replanning, all branches for planned changes need to be kept in sync by repeated merging, which may require manual resolution of merging conflicts. Thus, we require uniform language concepts that enable planning of all SPL artifacts in concert with feature models.

Multiple approaches exist that can capture the evolution of artifacts in general. Most of these approaches capture artifact evolution in terms of model evolution operations that are derived by computing differences between two model versions [KHH+09, NNP+10, EVC+07, EHK+02, KKT13, HK10]. Additionally, approaches exist that capture artifact evolution in terms of delta operations [CHS11, SBB+10, SSA14a, LKS16]. The aforementioned approaches rely on (delta) operations and, thus, operations must be explicitly defined to enable model modifications. Consequently, for each artifact language, a respective operation language must be devised. As a multitude of different artifacts and respective languages may exist in an SPL, such as feature models, implementation artifacts, documentation, and feature-artifact mapping, many operation languages must be defined. Each of those operation languages differs and is artifact-specific. Consequently, no uniform language to capture artifact evolution exists which makes it hard to model the evolution of different SPL artifacts. Moreover, similar to operation-based approaches for feature models, it is complicated to retrieve a particular artifact version as all operations until that version have to be applied each time a version is retrieved. Gîrba et al. [GD06] introduce *Hismo* that introduces first-class entities to model evolution. Each model element is extended by versions and states. However, this method introduces an entirely new modeling language that is not compatible with the original models and tool infrastructure. Thus, practical applicability is not given. Moreover, none of the previously mentioned approaches provides concepts for the planning of SPL evolution.

Challenge 4: Uniform Modeling of SPL Artifact Evolution

Uniform modeling methods that enable capturing and planning of evolution of all SPL artifacts in concert with feature models are required.

Updating a feature model and feature-artifact mappings may result in changed product behavior as existing configurations may use different implementation artifacts than before evolution. In general, it is desirable to preserve product behavior but, in some cases, this is not possible, for instance, if a feature has been deleted or has got reduced functionality. In other cases, features may be split resulting in more fine-grained variability, which can be used to remove unused functionality in products. Thus, engineers need to evolve configurations in concert with feature models and feature-artifact mappings to preserve product behavior or to make an informed decision on how product behavior is changed. Existing approaches either ignore that mappings evolve and repair only invalid configurations [WSB+08, WPX+13, XHS+12], or allow only behavior-preserving refactorings limiting SPL evolution [SBT16, STK+12, NBA+15]. Additionally, all these approaches assume that the engineer maintaining configurations exactly knows in which way the SPL evolved.

However, for large or open-source SPLs, such as the Linux kernel, *domain engineers* specify feature models and feature-artifact mappings, and *application engineers* create and maintain configurations. Thus, domain engineers know which parts of the SPL evolved in which way, and application engineers know the requirements of the deployed products and can decide which changes to product behavior are admissible. Without knowledge of both domain engineers and application engineers, the evolution of feature models, feature-artifact mappings, and configurations may lead to altered product behavior, which may remain undetected resulting in unexpected behavior of deployed products. Thus, domain engineers need methods to express how they modified parts of the feature model and the feature-artifact mapping, how this impacts configurations, and how configurations could be updated. Using this information, application engineers need to be able to make informed decisions on how to update their configurations.

Some research focuses on fixing invalid configurations. White et al. present an automatic approach that computes the smallest possible set of changes in the configuration to fix it [WSB+08]. Xiong et al. propose a semi-automatic approach that provides the complete set of fixes with the smallest number of feature changes [XHS+12], whereas the approach by Wang et al. gradually reaches the desired fix using application engineers' feedback [WPX+13]. Both semi-automatic approaches assume that the person fixing the configuration knows what the best fix is. Moreover, these approaches do not take the product behavior of a configuration into account. Thus, fixes may lead to different product behavior and, therefore, provide a false sense of correctness. Multiple other approaches also consider product behavior. Borba et al. devised a refinement theory for SPL evolution preserving product behavior [BTG12]. Neves et al. proposed several evolution templates preserving product behavior using this theory [NBA+15]. Sampaio et al. extended this theory by introducing partially safe evolution templates, preserving product behavior for a subset of configurations [SBT16]. However, these approaches do not provide support for configuration update operations even if product behavior is not preserved. Additionally, they do not consider the communication barrier between domain engineers and application engineers.

Challenge 5: Updating Configurations after SPL Evolution

A method for updating configurations in concert with feature model and feature-artifact mapping evolution that enables domain engineers to share their knowledge on evolution with application engineers is required.

1.2. Research Questions

With state-of-the-art methods, it is not possible to model SPL evolution and solve the challenges mentioned above. Thus, we identified this as a research gap and will address the described challenges in this thesis. This leads to our main research question we want to answer in this thesis:

Main Research Question

How can we track, execute, and plan feature-model evolution to drive consistent SPL evolution?

We divide this overall research question into three sub-research questions that target different parts of the overall question in more detail.

Research Question RQ1 – Modeling the Entire Feature-Model Evolution Timeline. *How can we track, execute, plan, and replan feature-model evolution?* Within this research question, we want to investigate how to model an entire feature-model evolution timeline that keeps relations between evolution steps. Moreover, in such a timeline, planned future evolution becomes present and, afterwards past history. This research question addresses **Challenge 1: Modeling Feature-Model Evolution Timelines**. In particular, we want to find out how integrated modeling of evolution as first-class citizen in feature models can be realized that enables to capture the past history and plan future evolution with the possibility of replanning. To provide the basis for further analyses, we investigate how to preserve information on which elements evolved in which way. Moreover, we want to determine tool support for modeling and showing this evolution as this is crucial to manage the additional complexity introduced by evolution.

Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention. *How can we preserve feature-model consistency in presence of replanning and keep feature-model evolution timelines free from anomalies by incorporating information on evolution in explanations?* Within this research question, we investigate how to prevent the introduction of feature-model inconsistencies when changing the evolution timeline which addresses **Challenge 2: Consistent Feature-Model Evolution Replanning**. To this end, we need to find out how analyses can detect whether an evolution operation would introduce an inconsistency in order to prevent the evolution operation's execution. Additionally, we investigate how to detect and resolve feature-model anomalies in the evolution timeline. As feature-model evolution timelines are complex, we want to analyze how to provide more meaningful explanations that incorporate evolution information. Methods for detection and explanation of anomalies address **Challenge 3: Detecting and Explaining Anomalies in Feature-Model Timelines**. As it is important to prevent inconsistencies, and detect and explain anomalies when devising feature models in practice, we want to find out how to integrate this in tool support for modeling feature-model evolution timelines (**RQ1**).

Research Question RQ3 – Consistent SPL Artifact Evolution. *How can we enable consistent evolution of SPL artifacts consisting of feature models, realization artifacts, feature-artifact mapping, and configurations?* We want to find uniform language concepts that enable modeling of SPL artifact evolution, such as realization artifacts, feature-artifact mappings, or configurations, in concert with feature-model evolution, which addresses **Challenge 4: Uniform Modeling of SPL Artifact Evolution**. Such a uniform language concept to capture evolution forms the basis to keep all SPL artifacts consistent, i.e., in a compatible state. As myriads of artifact languages may exist, it is infeasible to manually adapt existing languages with the evolution language concepts. Thus, we investigate how generic methods can be provided that automatically extends artifact languages by evolution while preserving compatibility with existing tools. Keeping configurations consistent with SPL version requires updating configurations. This requires knowledge of both, domain and application engineers, and, thus, we investigate how knowledge can be shared between those engineers. Methods that overcome the communication barrier between domain and application engineers need also to consider the fact that updating configurations may occur time independent from feature-model evolution. Moreover, we want to find out how sharing this knowledge can be used to preserve the product behavior of existing configurations after evolution. To increase applicability in practice, we investigate how

tool support can be devised that provides a high automation degree. The methods to update configurations address **Challenge 5: Updating Configurations after SPL Evolution**.

1.3. Contributions and Structure of this Thesis

In this thesis, we provide methods for consistent integrated modeling and consistency analyses of SPL evolution timelines. In Chapter 2, we provide foundations that are needed to understand this thesis. The contributions of this thesis can be grouped into three main areas. Figure 1.1 shows the overview of our contributions. The three areas of contributions are structured following the research questions **RQ1 – RQ3**.

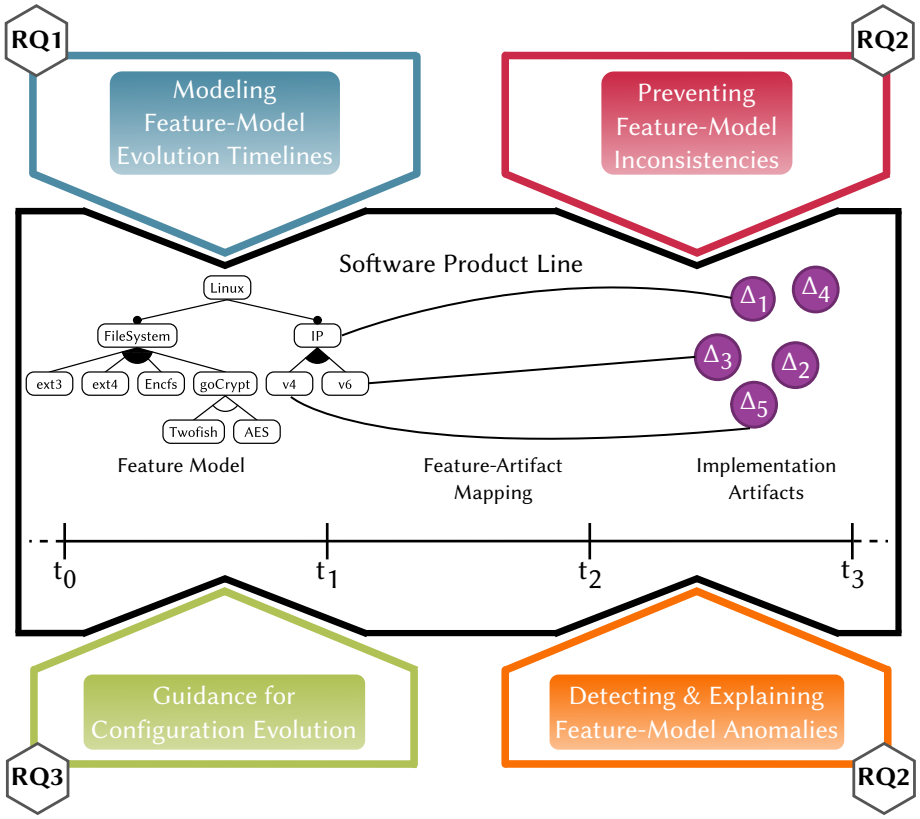


Figure 1.1.: Overview of the Contributions

Part II – Modeling Feature-Model Evolution Timelines presents a modeling notation called Temporal Feature Models (TFMs) that introduces evolution as first-class citizen. With TFMs, engineers are able to capture the entire evolution timeline of a feature model in one model. They are able to track exactly which feature-model elements evolve in which way. In addition, SPL evolution can be planned by modeling future states of the feature model while modifying the current status in parallel. With this contribution, we address **Challenge 1: Modeling Feature-Model Evolution Timelines**.

Part III – Analyzing Feature-Model Evolution Timelines deals with analyses of feature-model evolution timelines which can be used by engineers to ensure consistency of the evolution and keep the feature-model evolution timeline anomaly-free. In Chapter 4, we provide a method to detect whether changed or newly introduced evolution operations violate the consistency of other already

planned evolution steps. With this contribution, we target **Challenge 2: Consistent Feature-Model Evolution Replanning**. Moreover, we provide a method to search for anomalies in the entire evolution timeline of a feature model in Chapter 5. We explain these anomalies by identifying causing evolution operations. Compared to state-of-the-art explanations, these explanations have a reduced complexity and yet are more meaningful. This contribution addresses **Challenge 3: Detecting and Explaining Anomalies in Feature-Model Timelines**.

Part IV – Consistent Software Product Line Artifact Evolution provides contributions for consistent evolution of SPL artifacts. In Chapter 6, we generalize the concept to model evolution of TFMs for arbitrary (meta-) models that allows us to capture the evolution timeline of other SPL artifacts, i.e., implementation models, feature-artifact mappings, and configurations. To this end, we provide an automatic generation process that augments existing metamodels while it preserves compatibility to existing tooling using an adapter infrastructure. This contribution addresses **Challenge 4: Uniform Modeling of SPL Artifact Evolution**. In Chapter 7, we provide a methodology to update configurations to new SPL versions. While we provide concepts for a uniform language to model the evolution of SPL artifacts in Chapter 6, we focus on the evolution of configurations in accordance with feature-model evolution. To this end, we provide a formal notation for domain engineers to express feature-model and feature-artifact mapping evolution. We enable domain engineers to transfer their knowledge about the evolution to application engineers by providing guidance on how to update existing configurations. This guidance can be used by application engineers to automatically update existing configurations. If an automatic update is not possible, application engineers can use the information provided by domain engineers to make an informed decision on how to manually update existing configurations. With this contribution, we target **Challenge 5: Updating Configurations after SPL Evolution**.

To make the developed concepts applicable, we also propose tool support. The tool suite integrating all previously mentioned methods and analyses is called DarwinSPL.[4] This tool suite is used in each chapter as basis for the presented evaluations of the respective contributions. We summarize this thesis in Chapter 8 with an outlook to further future research directions that are based on our contributions.

[4] https://gitlab.com/DarwinSPL/DarwinSPL

2 Background

In this chapter, we detail background that is relevant to understand the context and contribution of this thesis. First, we introduce variability management in terms of Software Product Lines (SPLs). Second, we elaborate on methods to analyze feature models. Third, we discuss multiple theories that we use in this thesis to reason on feature models. Fourth, we give an introduction to model-driven software development which we use as technical basis for all artifacts presented in this thesis.

2.1. Software Product Lines

Today's software systems are typically developed to address a wide range of user requirements [SRC+12]. At the same time, user customization plays a pivotal role for a software system to be successful [PBL05]. Users require a product that is custom-tailored to their needs and provides all required capabilities, but does not contain unnecessary functionality. As a result, many *variants* of a software system need to be developed in parallel [SRC+12]. This leads to further challenges as many related software systems are developed that share *commonalities*, but each also provides *variabilities* compared to the other systems. Without proper approaches, variants are cloned and modified to fit a specific user's needs. This process is called *clone-and-own* and is efficient as it can be performed without additional process overhead, or tools [DRB+13]. Over the course of time, many clones are developed that originally shared commonalities. As soon as changes to commonalities are required, a major disadvantage of clone-and-own becomes apparent: it is unclear to which other clones or variants such a change can be reintegrated as the code base may have diverged strongly and, moreover, merging such changes into all variants results in high effort. Apart from that, additional functionality that has been developed for a certain clone may be relevant to other clones as well, but is typically not integrated as this requires significant manual effort.

To overcome such an unstructured approach, SPLs are introduced to enable structured reuse and mass customization for families of software systems [CN01]. With SPLs, commonalities and variabilities are planned a priori in terms of *features*.

Definition 2.1: Feature

"A feature is a characteristic or end-user-visible behavior of a software system." [ABK+16]

An SPL allows to derive *variants* or *products* based on a feature selection that is used by a specific generative mechanism [CEC00]. For instance, the Linux kernel is one of the largest publicly available SPLs [SLB+10]. Depending on the version, it has more than 6,000 features [KTM+17]. Many very diverse systems base on the Linux kernel, such as smartphones[1], desktop computers or servers[2], car infotainment systems[3], or internet routers[4]. These diverse systems emphasize on the benefits

[1] https://www.android.com
[2] https://ubuntu.com/
[3] https://www.automotivelinux.org/
[4] https://openwrt.org/

of SPLs. First, effort can be saved when developing variants as common artifacts can be reused. Only new or changed features have to be implemented for new variants. Additionally, this also reduces the time to create a new software product compared to implementing each variant individually. However, developing an SPL requires upfront effort as devising artifacts, such that they are reusable, is more complex than developing them for a single system. Nonetheless, if many variants exist, the effort saved when reusing artifacts exceeds the necessary upfront effort [BCM+04]. Second, maintaining the common code base requires less effort as commonalities are captured in terms of the same reused artifacts. Consequently, if bugs are fixed or features evolve, all variants that share the affected features can directly benefit from the changes. In contrast, for variants created using a clone-and-own approach, such changes need to be merged into each variant individually.

Domain Engineering Developing an SPL is a complex endeavor and, thus, a suitable engineering process is required. The classic SPL engineering distinguishes between *domain engineering* and *application engineering* [PBL05]. During domain engineering, engineers specify a roadmap of products that should be supported by the SPL. This roadmap is used to define domain requirements which contain common as well as product-specific requirements. The output of the domain requirements engineering phase is the *problem space* of the SPL that contains abstract descriptions of the provided functionalities in terms of features. Typically, features are captured in a *variability model* that describes the relations and dependencies between features.

In the domain realization phase, the *solution space* is defined that contains the artifacts realizing concrete functionality. This comprises code, design models, documentation, or tests and typically requires explicit language concepts to realize variability. The *configuration knowledge* or *feature-artifact mapping* connects the problem and solution space [CEC00]. In particular, the feature-artifact mapping describes which realization artifacts are used for certain feature combinations.

Application Engineering The goal of the application engineering is to derive a final product from the SPL [PBL05]. To this end, application engineers collect requirements from customers or end users that describe the desired product. Using these requirements, application engineers define a *configuration*. A configuration is a set of selected features. Additionally, a configuration may also contain explicitly deselected features. A configuration is *valid* if its feature (de)selection satisfies all constraints imposed by the variability model. Based on a configuration, all required realization artifacts are derived using the feature-artifact mapping. This step is typically performed automatically as well as the subsequent *product generation* that creates a product based on the derived realization artifacts [CEC00].

2.1.1. Feature Models

The variability model of an SPL is a central artifact that is used for planning of the SPL, communication between engineers and management, and by customers to select the functionality of their desired product. The most common variability model type are feature models [BRN+13]. A feature model structures the features in a tree-like hierarchy [KCH+90, CEC00]. It has exactly one root feature and each other feature has exactly one parent feature. Each feature has a type: a MANDATORY feature *must* be selected if its parent feature is selected, whereas an OPTIONAL feature *can* be selected if its parent feature is selected. The root feature is always MANDATORY. Additionally, features are organized in groups and each group has a type: an AND group is a collection of OPTIONAL and MANDATORY features and does not pose additional constraints; if a parent feature of an OR group is

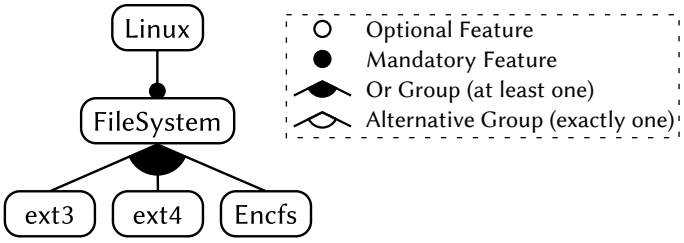


Figure 2.1.: Exemplary and Simplified Feature Model of the Linux Kernel.

selected, at least one feature of that group has to be selected as well; in an ALTERNATIVE group, exactly one feature has to be selected if the group's parent feature is selected.

A feature diagram is a visualization of a feature model [KCH+90]. For instance, Figure 2.1 shows a simplified excerpt of the Linux kernel feature model. It shows features that are responsible to provide support for different file systems. The features `ext3` and `ext4` are file systems that are typically used (or have been used) as default file system for many Linux distributions. The `Encfs` feature is an extension that can be used to encrypt files. Under the root feature `Linux`, a MANDATORY feature `FileSystem` is contained in an AND group that is not explicitly visualized. The features `ext3`, `ext4`, and `Encfs` are contained in an OR group under `FileSystem` and represent different concrete manifestations of the feature `FileSystem`. A valid configuration of that feature model always needs to contain the features `Linux` and `FileSystem`, and contains at least one of the other file system features.

Not all relations between features can be expressed using the tree hierarchy or types of a feature model. To this end, *cross-tree constraints* can be defined which specify relations between features. *Simple* cross-tree constraints define *requires* and *excludes* relations between features, whereas *complex* cross-tree constraints use propositional logic with features as literals [KTM+17].

Several extensions to feature models exist. Cardinality-based feature models use multiplicities instead of feature and group types [CHE05]. The goal of cardinality-based feature models is to be more flexible regarding the constraints imposed by feature and group types, and to allow multiple instantiations of a feature. In particular for cyber-physical systems, the number of certain subsystems or components plays a pivotal role. Each feature and group is associated with a specific cardinality that defines the lower and the upper bound of the number of the feature's instantiations. For a group, the cardinality specifies how many features of that group can or must be selected.

To express more fine-grained variability, *feature attributes* have been introduced [BTR05]. A feature attribute is assigned to a feature and has a type. Typical types are numbers, strings, enumerations, or Booleans. For instance, an attribute `key strength` of the feature `Encfs` (cf. Figure 2.1) could be used to further specify the encryption strength of that file system. However, it is also possible to define static attributes, such as a price or memory consumption of features. Such attributes can then be used to optimize configurations, e.g., to generate the cheapest product.

2.1.2. Variability Realization Mechanisms

After defining variability on an abstract level in a variability model, artifacts that realize the concrete products need to be implemented. However, in contrast to single software development, explicit mechanisms to express variability in implementation artifacts are required. To this end, several approaches exist [SRC+12, KAK08, CA05, Bat04]. In *annotative* or *negative* vari-

ability realization mechanisms, parts of artifacts are annotated with *presence conditions*. A presence condition defines for which feature selections the annotated part is present in a product. All other parts are removed during product generation. A prominent example of annotative approaches are C/C++ preprocessor directives.

In *compositional* or *positive* variability mechanisms, additional functionality is added to a product, based on the feature selection. In contrast to annotative approaches, in compositional approaches, the linking between variable artifact (parts) are not directly modeled in the realization artifacts themselves, but has to be defined explicitly. Upon variant generation, all components that are mapped to the selected features are collected and are composed with a core artifact. A typical example of a compositional approach is *Feature-Oriented Programming* (FOP) [Bat04], but also plug-in or component architectures lean themselves for this approach.

Finally, in *transformational* approaches, it is possible to add, delete, or modify artifact parts. A potentially empty base variant serves as basis to which transformation operations are applied. Similar to compositional variability, an explicit mapping from features to transformation operations is required. To derive a variant, all operations that are mapped to the selected features are applied to the base variant. Typically, a partial order between those operations exists. Delta-Oriented Programming (DOP) [SBB+10, SD10, CHS11] is a prominent example of this approach. In DOP, *delta operations* are responsible for transforming a base variant and the delta operations are structured in *delta modules* which are mapped to features.

2.2. Feature Model Analyses

An SPL is a complex family of software systems that needs to incorporate many requirements from different stakeholders. The feature model serves as a main communication artifact as it describes an SPL's functionality on an abstract level without technical details. Thus, SPL development optimally starts with the feature model [PBL05, PCA+13]. For a frictionless SPL development process, the feature model must be consistent and without anomalies. To this end, several analyses exist that reason on feature models [SRC+12, BSR10, BTR05, GBT+]. In this section, we present two different fields of feature-model analyses: structural inconsistency and configuration logic anomalies.

2.2.1. Well-Formedness

Much research considered well-formedness rules of feature models that ensure its structural consistency [GMB06, CW07, BSR10, SHT+07, JK07, SHT06]. Based on the well-formedness rules defined in the literature, we extract those that are relevant for our notion of feature models. In the following, we provide a brief informal description of each well-formedness rule.

WF1 The root feature must be part of each configuration and, thus, it must be of type MANDATORY.

WF2 Each feature must be identified unambiguously. Consequently, each feature name must be unique. Thus, we define that the names of all pairwise features must be different.

WF3 To ensure that a feature is contained in the tree, it must be part of a group. Additionally, it may not be part of multiple groups as the position of a feature in the tree would then be ambiguous. In the formalism, we define that a feature is either the root feature or that it is contained in exactly one relation from groups to sub features.

WF4 Similar to features, a group must be a child of exactly one feature to define its position in the tree. To ensure this, we define that the number of relations from features to sub groups is exactly one for each group.

WF5 In ALTERNATIVE and OR groups, at least one feature needs to be selected. Thus, groups having such a type only make sense if they contain at least two features. Otherwise, a single feature would always have to be selected and, thus, should be modeled as MANDATORY. Consequently, we verify for each group that it is either of type AND or that it is related to at least two sub features.

WF6 Similarly to the previous rule, a MANDATORY feature in an ALTERNATIVE or OR group does not make sense. As in an ALTERNATIVE, group exactly one feature must be selected, only the MANDATORY feature could be selected and all others can never be selected. Moreover, multiple MANDATORY features in an ALTERNATIVE group result in an unconfigurable and, thus, inconsistent feature model. For OR groups, MANDATORY features would result in misleading modeling as this is similar to an AND group with the MANDATORY features and the remaining OPTIONAL features. We define the well-formedness rule such that we verify for each group that it is either of type AND or that each of the features contained in the relation from the considered group has the type OPTIONAL.

If a well-formedness rule is violated and subsequent modifications base on the inconsistent feature model state, the feature model may become useless. Consequently, tools and methods modifying a feature model must ensure that all of these well-formedness rules hold to retrieve a consistent feature model.

2.2.2. Feature-Model Anomalies

Apart from structural well-formedness, different analyses deal with analyzing the configuration logic of a feature model [SRC+12, BSR10, BTR05, GBT+]. For most analyses, the feature model is translated into a propositional formula with features as literals [Bat05], e.g., into a Conjunctive Normal Form (CNF). The most evident analysis is whether a given configuration is valid, i.e., it fulfills all constraints imposed by the feature model. To this end, the selected and deselected features are used to set the respective feature literals to *true* or to *false*, respectively. Subsequently, it is validated whether this formula is satisfied, e.g., with a simple Boolean evaluation algorithm or a SAT solver. Another related analysis is whether a given *partial* configuration can still become valid [KTS+18]. In a partial configuration, some features are not part of the configuration, i.e., they can still be selected or deselected. A partial configuration can still become valid if selecting or deselecting features that are not part of the configuration can result in a valid configuration.

Feature models can become very large with many cross-tree constraints. For instance, the Linux kernel contains more than 6,000 features and more than 3,000 cross-tree constraints [KTM+17]. As a consequence, domain engineers are faced with the complex task to not create constraints that lead to unintended design. Feature-model *anomalies* may be the result of such an unintended design [BSR10]. Much like code smells, most feature-model anomalies may be an indicator for bad design choices and errors. The most prominent anomalies are:

- *Void feature model anomaly:* This anomaly arises if constraints contradict each other and no configuration exists that can fulfill the feature-model's constraints. As a consequence, no valid product can be generated anymore which renders the SPL useless.
- *Dead feature anomaly:* A feature is dead if it cannot be selected in any valid configuration. This anomaly is similar to dead code and results in a useless feature. Moreover, if a dead feature is actively maintained as engineers do not know that this feature is dead, this results in unnecessary effort.
- *False-optional feature anomaly:* A feature is false-optional if its type is OPTIONAL but it is part of each valid configuration in which its parent feature is included. In fact, it behaves like a MANDATORY feature, the OPTIONAL type may mislead engineers, and less configuration options as intended may be present.

Several approaches exist to detect feature model anomalies [KAT16, Hem08a, ML04, Bat05, LSW15, TBR+06, Tri12, FBG+13, KSR13]. Typically, an anomaly is detected by trying to find a solution for the propositional formula of the feature model in conjunction with additional formulas. For instance, to check whether a feature is dead, the respective feature literal is added as conjunction to the feature-model formula and a solver tries to find a solution for that combined formula. If it cannot find a solution, the feature is dead.

Fixing anomalies is well-advised as they may lead to errors and may propagate through feature-model evolution. However, finding the cause for an anomaly is a challenging task. To provide remedy, multiple methods provide explanations for anomalies. Typically, such an explanation consists of clauses of the feature-model formula that could not be satisfied [FBG+13, KAT16]. However, within the tool FeatureIDE, sophisticated support for anomaly explanation is implemented, highlighting structures in the feature diagram that are part of an explanation [AKT+16].

2.3. Theories for Reasoning on Feature Models

Analyzing feature models requires theories that are highly optimized to be applicable to large real-world SPLs. To this end, we first introduce theories to solve satisfiability problems which are used for many feature-model analyses, such as the detection of anomalies. Additionally, we give a brief introduction on structural operational semantics (SOS) which we will use in Chapter 4 to ensure feature-model well-formedness in presence of evolution.

2.3.1. Satisfiability Theories

Satisfiability problems deal with finding a solution for a given formula. The most common problem are SAT problems which consider the satisfiability of a propositional formula. A propositional formula ϕ holds a set of variables $var(\phi)$ and consists of literals (i.e., variables, *true*, or *false*) that are connected using logical operators (i.e., $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\leftrightarrow$ (bi-implication), $\neg$ (not)) [BSR10]. A truth value can be assigned to each variable, i.e., *true*, or *false*. For feature models, features are variables and a feature selection results in a *true* value, and a deselection in a *false* value, respectively.

SAT problems are NP-complete [NOT06, Joh92] and, thus, no deterministic algorithm can find a solution within polynomial time (if $P \neq NP$). As a result, feature-model analyses that base on SAT problems, e.g., feature-model anomaly analyses, are not trivial and result in high computation

times for large feature models. Satisfiability modulo theories (SMT) is trying to check the satisfiability of logical formulas using multiple theories [BHM09]. Modern SMT solvers are able to deal with first-order logic formulas and arithmetic operators.

2.3.2. Operational Logics

Feature-model well-formedness is crucial to keep the SPL functional. After each feature-model modification, the well-formedness must be ensured. One approach to reason on the impact of operations are *structural operational semantics (SOS)*, also called small-step semantics. With SOS it can be described how individual steps of a computation are executed [AFV01, Plo04]. An SOS specification is a set of inference rules, which can be defined in the following form:

$$\frac{Conditions}{State \Rightarrow State'}$$

An SOS rule describes a transition from *State* to $State'$ if all *Conditions* are satisfied. This way, it is possible to define SOS rules for feature-model operations that ensure with the *Conditions* that an operation can only be executed if it does not violate well-formedness rules.

Rewriting logic can be used to reason on computational and logical transitions of a system [Mes12, MM02, Mes00]. Much like SOS, rewriting logic consists of rewrite rules that define transitions from one state to another. A system that is represented in rewriting logic consists of the formalized system's state and a set of rewriting rule instantiations. Thus, SOS rules can be modeled as rewriting rules and a system's behavior or modifications can be analyzed using instantiations of those rules. *Maude* is a language and system that integrates modules in rewriting logic [CDE+07, Mes00]. It provides basic modules and can be extended by custom modules. Such custom modules can be used to define own rewriting rules. The Maude interpreter can be used to apply and verify rewriting rules.

2.4. Model-Driven Software Development

In model-driven software development, *models* are not only used for documentation purposes but play a pivotal role throughout the entire lifecycle of a project, e.g., for management, design, implementation, or behavioral specification. Typically, models are then used to automatically generate source code that can be directly deployed such that engineers do not need to write most parts of the code. In general, a model is an abstract representation of a system [VSB+13]. In particular, it represents the relevant parts of a considered system. Based on the use case, this may comprise the structure, the function, and the behavior.

A *metamodel* specifies the basic constructs that can be used to define a model. In particular, it defines the modeling notation in a structured way [Gro16]. As a result, each model is an instance of its metamodel. The Object Management Group (OMG) standardizes these concepts in the Meta Object Facility (MOF) [Gro16]. The MOF defines four layers: the real-world system is described in the layer M0; the model that is an abstraction of the real-world system of M0 is defined in the layer M1; the layer M2 contains the metamodel that specifies how models in M1 can be defined; finally, the meta-metamodel in the layer M3 is defined in the MOF and describes concepts how to define metamodels. A special characteristic of the meta-metamodel is that it is defined using the concepts defined in the meta-metamodel itself and, thus, represents the "end" of the layers.

The meta-metamodel describes, inter alia, classes, attributes and references between classes. A reference describes relations between classes and has a multiplicity denoting the lower and upper bounds on the number of referenced class instances. A containment reference is a special type of reference, which denotes that the containing class is owning the contained class, i.e., class instances of the contained class only exist within the lifespan of class instances of the containing class. An attribute defines additional properties of a class and is similar to a reference but, instead of referencing objects of another class, it stores values of a primitive type. For instance, in a metamodel for feature models, a class for features and a class for groups would be defined. References between the feature class and the group class are used to describe the tree hierarchy, and attributes for the feature class could be used to describe a feature's type or name.

A model is then a concrete instantiation of the metamodel and contains instances of the classes (referred to as objects), references, and attributes defined in the metamodel. For instance, a concrete feature model such as the Linux kernel feature model with multiple feature and group objects. The *Ecore* meta-metamodel of the *Eclipse Modeling Framework (EMF)*[5] is an implementation of the most essential elements of the MOF meta-metamodel. With corresponding tools and extensions, it allows to define, and use metamodels and models in Eclipse[6].

[5] https://www.eclipse.org/modeling/emf/
[6] https://www.eclipse.org/

Part II.

Modeling Feature-Model Evolution Timelines

3 Tracking and Planning of Feature-Model Evolution

The contents of this chapter are largely based on the work published in [NSS16, NES17, HNL+19].

Summary *Capturing and planning feature-model evolution is crucial to analyze the evolution timeline and to thoroughly plan the future evolution of an SPL. Existing technologies, such as Version Control Systems (VCSs), and feature-model evolution notations do not provide the necessary preciseness or expressiveness to adequately model entire feature-model timelines. Thus, reasoning is based on approximations performed by differencing mechanisms and engineers are not able to model future evolution. In this chapter, we present TFMs – a novel notation that is capable to capture past evolution, the present state, and future evolution of a feature model in one artifact. We put a particular focus on future planning and enable engineers to introduce intermediate evolution steps that realize unplannable ad-hoc changes or incrementally detail the evolution plan. Using these contributions, engineers can use TFMs as living artifact: as time passes this future becomes present and, afterward, past which allows to reason on the entire evolution timeline.*

Software evolution and maintenance are some of the most costly and time-consuming tasks when developing software and, thus, they are key disciplines in software engineering [Boe84]. For long-living software systems, these tasks are even more crucial as the initial development phase is done once and, afterward, evolution and maintenance are the main activity. Typically, a Software Product Line (SPL) is an especially long-living software system as the initial development phase is very expensive and, thus, the SPL has to exist for a long time to be profitable [BPD+10, BP14, BCM+04, FK05]. Additionally, the evolution of an SPL is more complex than the evolution of individual software systems as many products can be derived with different requirements and many involved stakeholders [BCM+04, WGS+14, BP14]. Thus, when performing SPL evolution, the impact of the evolution on all existing products and the introduction of new products has to be considered [Liv11, BP14]. Furthermore, an SPL is typically embedded in other company processes, e.g., for car manufacturers, different other processes correlate with SPL evolution such as hardware systems, production processes, or marketing. Consequently, SPL evolution is generally not performed in an ad-hoc manner but must be thoroughly planned and correlated processes need to be adapted subsequently [BPD+10, EBL+10, WGS+14]. This enables engineers to define milestones for evolution, monitor the evolution progress, and to perform analyses for the planned state after evolution.

Optimally, SPL evolution starts with the feature model as this is the main communication artifact for all stakeholders and represents the entire functionality of an SPL on a conceptual level [PCA+13]. Planning SPL evolution by modeling these in terms of future feature model versions is especially suitable as the features describe the conceptual part of an SPL without the need for implementing the actual functionality [PBD+12]. Additionally, abstraction on a

feature-model level can be used to synchronize and evolve other company processes as well, such as manufacturing, logistics, or marketing.

Enacting planned SPL evolution is an incremental process as the planned features cannot be implemented all at once. Consequently, intermediate evolution steps must be introduced which incrementally realize features of the plan. However, planned evolution rarely goes as intended and, consequently, unplanned changes have to be introduced in intermediate steps as well [WGS+14, BP14]. The probability of these divergences increases the longer the evolution is planned. Additionally, changes on short notice have to be incorporated as well. For instance, if an important stakeholder has last-minute changes in requirements or if legislation changes unforeseen.

Learning from past evolution is crucial to identify sources of problems, to improve development processes, to identify and predict statistical trends of the SPL evolution, and to estimate impact of changes based on similar previous changes [Liv11]. Analyses for past evolution can be supplemented by incorporating information on planned evolution. For instance, the growth of the complexity of an SPL in terms of available configurations can be retrieved by analyzing the past feature-model evolution history. These results can be used to predict future growth as well. By comparing to or incorporating the planned feature-model evolution, engineers and management can set goals for a target growth or can retrieve more precise predictions. In summary, history and planned evolution of a feature model must be captured to learn from history, to enable thorough planning, to enable the incremental realization of the planned evolution, and to set the basis for expressive analyses.

In this chapter, we introduce Temporal Feature Models (TFMs), a concept for integrated capturing of the entire evolution timeline, i.e., past history and planned evolution, of a feature model in one artifact. Figure 3.1 shows the overview of our contributions in this thesis. Modeling TFMs forms the basis to perform and plan SPL evolution and with this contribution, we address **Challenge 1: Modeling Feature-Model Evolution Timelines** and answer **Research Question RQ1 – Modeling the Entire Feature-Model Evolution Timeline**.

This chapter is structured as follows: first, we define a typical evolution use case and derive requirements for a feature-model evolution notation in Section 3.1. In Section 3.2, we provide a metamodel and a formalism for TFMs. In Section 3.3, we evaluate our methods by showing feasibility in terms of implementation and application to real-world feature-model evolution. In Section 3.4, we give an overview of other concepts to capture feature-model evolution, compare them with our method, and identify other related work. Finally, we close this chapter with a summary in Section 3.5.

3.1. Requirements for Capturing Feature-Model Evolution

Reasoning on and planning of feature-model evolution requires a modeling notation that can capture the entire feature-model evolution timeline, i.e., the history and the planned evolution. However, typically plans never go as intended. This has multiple reasons. First, changes that are planned for future points in time are coarse-grained as not all side conditions are known. Consequently, these plans are detailed as time passes. However, detailing plans may reveal infeasible parts of a plan. Second, changes on short notice are common practice and require to change the original plans or shift changes to other points in time. Thus, a modeling notation to capture plans must also enable to replan and to introduce intermediate evolution steps. To provide these capabilities, concrete knowledge of the temporal relations between the evolution steps is required.

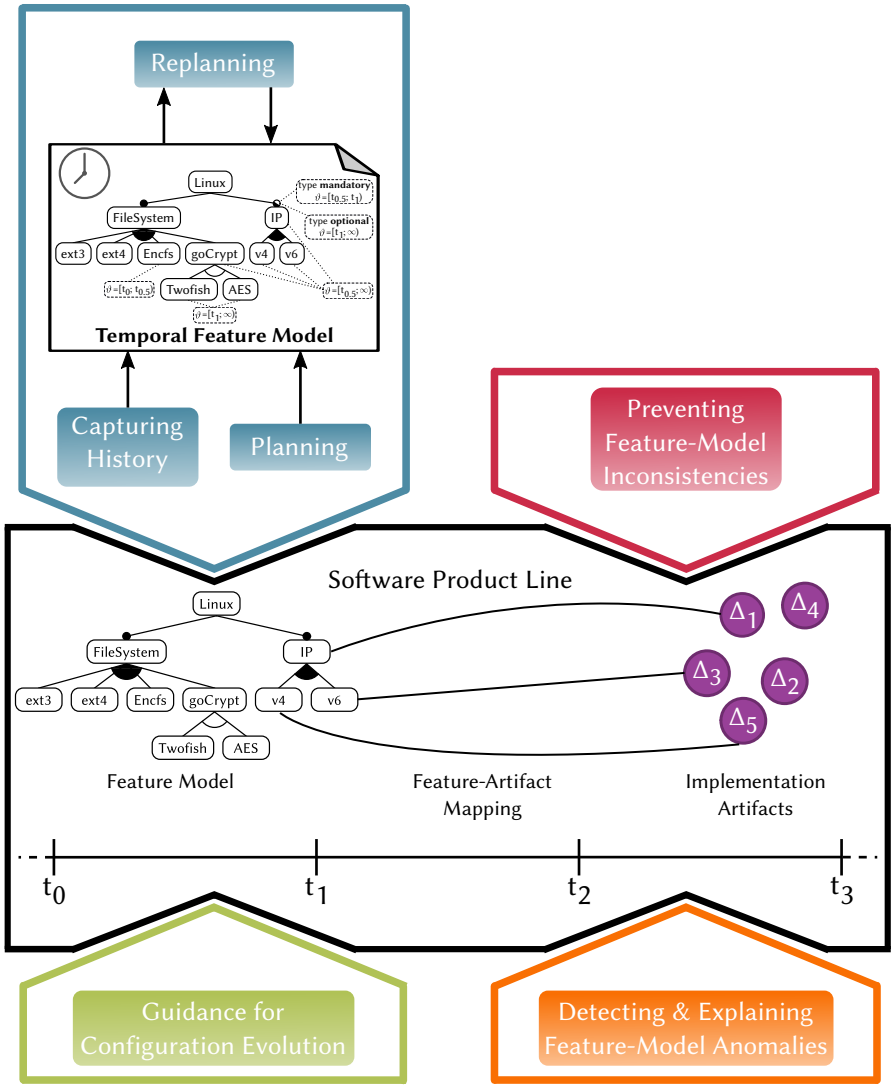


Figure 3.1.: Contribution Overview – Step 1, Modeling and Planning Feature-Model Evolution

In summary, tracking, planning, and replanning of feature model evolution require models specifically tailored to capture the evolution of all feature-model elements, their temporal relation, and the possibility to introduce evolution steps for any point in time. We consider points in time as symbolic points with an order, i.e., revisions, or even as real points in time, i.e., dates. Thus, we need a modeling concept that goes beyond the state of the art methods to capture evolution. These methods are not suitable for planning and analyzing feature-model evolution for two main reasons. First, planning of future evolution is not explicitly addressed and, thus, changing plans or introducing intermediate steps is not possible without unsuitable workarounds such as branching. Second, analyses of the feature-model evolution are only possible by computing model differences of different snapshots which is costly and inaccurate. Thus, we need a concept where evolution is stored on the model-element level and the temporal relation between evolution steps is explicitly modeled.

To derive requirements for a notation to track and plan feature-model evolution, we use the simplified Linux kernel feature model of Figure 2.1 (cf. Section 2.1) and extend it with an evolution use case. Figure 3.2 shows the initial version of the feature model for t_0, which represents the current

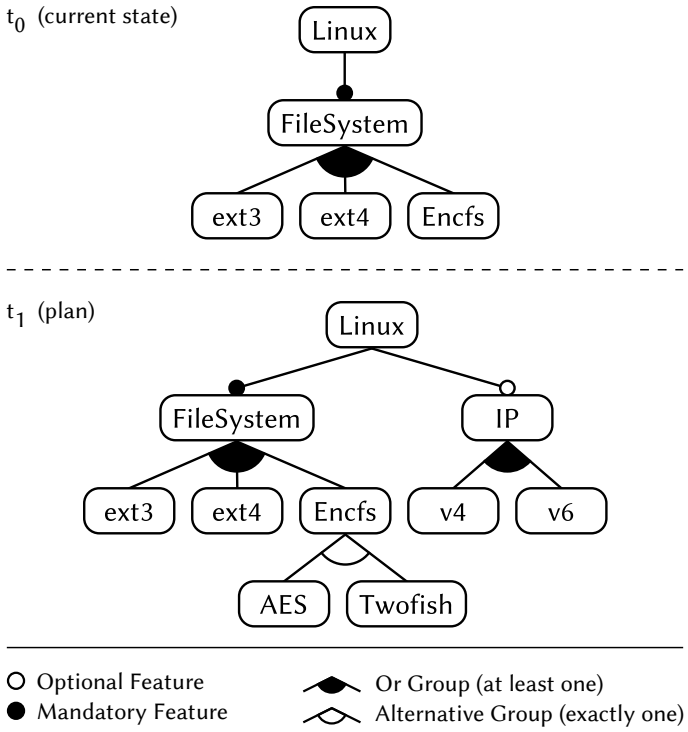


Figure 3.2.: Part of the Linux kernel feature model and its planned evolution.

state in the example. Additionally, the evolution of the feature model is planned for t_1. In this plan, two different encryption ciphers for the feature `Encfs` are added in an ALTERNATIVE group under `Encfs`. Additionally, an OPTIONAL `IP` feature is introduced which supports IP `v4`, IP `v6`, or both.

The planned changes for t_1 are incrementally realized in an agile manner. However, plans are most of the time realized differently than originally planned. To realize first changes planned for t_1 and to accommodate for change requests on short notice, an intermediate evolution step at $t_{0.5}$ is introduced. Figure 3.3 shows the adapted evolution scenario. Originally, the `IP` feature was planned to be OPTIONAL (cf. Figure 3.2) but most devices require IP-based network connections, engineers decide to realize this as MANDATORY feature. Additionally, the `Encfs` became outdated and a new file system feature (`goCrypt`) supporting encryption is introduced as successor. Consequently, `Encfs` is removed at $t_{0.5}$.

The changes introduced in the intermediate step also affect the original plan for t_1. The type of the IP feature should also be MANDATORY and not OPTIONAL. Additionally, the parent feature of the features `AES` and `Twofish` was `Encfs` in the original plan. However, `Encfs` was removed in the intermediate evolution stop. Consequently, `AES` and `Twofish` should be moved in a group beneath `goCrypt`. Finally, the feature model state at t_0 becomes history but should be kept for analyses and modeling purposes. For instance, to move `AES` and `Twofish` to `goCrypt` we need to know that their original parent has been deleted which exists only at t_0.

From this use case, we identify different requirements a modeling notation for feature model evolution needs to fulfill:

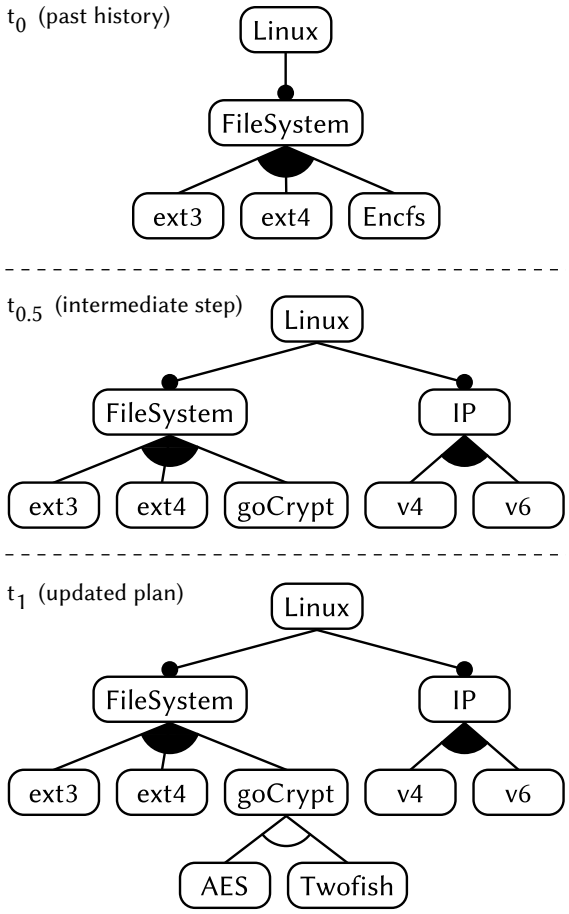


Figure 3.3.: Linux kernel feature model and its planned evolution with an intermediate evolution step at $t_{0.5}$.

Req.1: Adding and removing features and groups. It must be possible to introduce and remove features into a feature model. To create a tree hierarchy, it must be possible to create and remove groups as well.

Req.2: Changing feature and group types. If the relation between parent and child features changes, it must be possible to change the types of features and groups. For instance, if, originally, a child feature must always be selected if the parent feature is selected. After evolution, the child feature should also be deselectable if the parent feature is selected, the type of the child feature must be changed from MANDATORY to OPTIONAL.

Req.3: Changing feature names. Sometimes for diverse reasons, a feature name must be changed. For instance, if the marketing department demands a more appealing name, or if a feature technically significantly changed which should be reflected by its name.

Req.4: Changing the tree structure. After long evolution timelines, feature models significantly changed and, potentially have grown a lot. To prevent structural decay and to accommodate for new relations between features, it must be possible to move features and entire groups.

Req.5: Capturing past history. It is always important to learn from past events. This also includes feature models. Thus, to be able to analyze past feature model history, this history must be captured in an accessible way.

Req.6: Planning of future changes. As already pointed out, product-line evolution is a large endeavor and requires thorough planning. Thus, it must be possible to specify the future evolution steps of a feature model.

Req.7: Performing intermediate changes. Active development has to go on, even if plans have been made. This is necessary to account for short-notice changes, for detailing plans, and for unplannable changes. Thus, it must be possible to introduce evolution steps before and between already planned evolution steps.

Req.8: Capture temporal relation between evolution steps. To reason about consistency and to perform analyses, the temporal relation between evolution steps must be captured. For instance, it is possible to know that an intermediate step in which a feature is deleted is specified for an earlier point in time than a planned evolution step in which a new child feature is added to the respective feature.

Req.9: Direct retrieval of changes without additional computation. To perform analyses and to ensure consistency, changes between evolution steps must be evaluated. Long evolution histories result in many changes that must be analyzed. Thus, it is important that these changes can be directly retrieved and must not be computed for each analysis by comparing multiple feature-model versions. Consequently, these changes must be directly accessible without the need for additional computation.

In the example of Figures 3.2 and 3.3, no feature has been renamed or moved. Nonetheless, a modeling notation should explicitly capture this to support all possible change operations.

3.2. Temporal Feature Models

Integrated storage of history and planned evolution of feature models with the exact chronological relation of evolution steps, the possibility to add intermediate evolution steps, and without the drawback of approximations are necessary as outlined in the section before. To fulfill these requirements, we introduce the generic concept of *temporal elements* that forms the basis for storing a model timeline in one model artifact [NSS16].

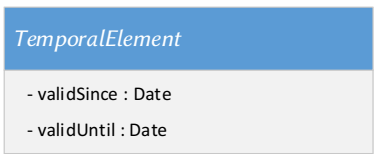


Figure 3.4.: Metamodel for Temporal Elements.

Figure 3.4 shows the metamodel for temporal elements. Each temporal element has a *temporal validity* $\vartheta = [\vartheta_{since}; \vartheta_{until})$ – a right-open interval that defines a timespan in which a respective element is valid. Thus, engineers can define for each model element when it starts to

be valid and when it ends to be valid. Planning evolution can be performed by setting the validity to a future point in time. With this modeling concept, differencing becomes obsolete as the exact differences are directly integrated into the model and the drawback of approximations by differencing mechanisms does not exist.

3.2.1. Temporal Feature Model Metamodel

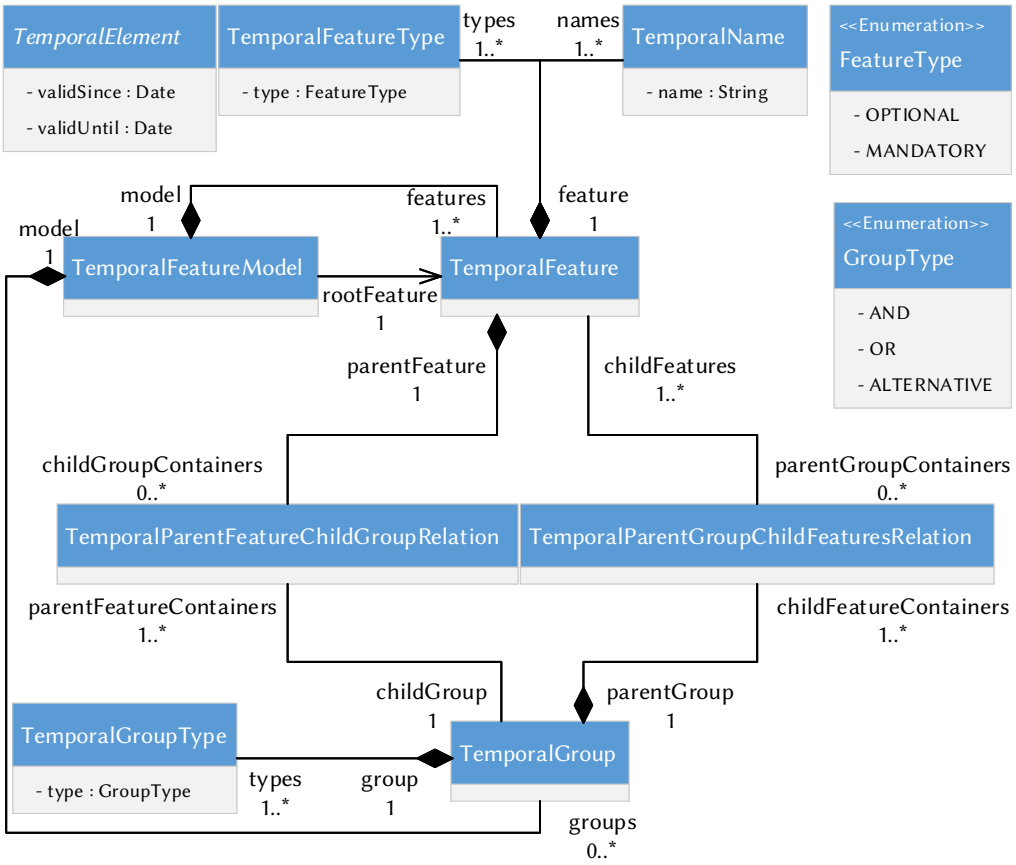


Figure 3.5.: Metamodel for Temporal Feature Models.

To enable precise and flexible tracking and planning of feature-model evolution, we apply the concept of *temporal elements* to all elements of a feature model for which we want to capture evolution [NSS16]. Figure 3.5 shows the metamodel of Temporal Feature Models (TFMs). Each entity in this metamodel except for `TemporalFeatureModel` inherits from `TemporalElement` – for brevity, we omitted the inheritance relations in the diagram. Consequently, we can store the evolution of each of those elements.

We explain this metamodel by following the containment hierarchy, starting with the `TemporalFeatureModel` that represents a TFM. Each `TemporalFeatureModel` contains a set of `TemporalFeatures` and a set of `TemporalGroups`. Moreover, the `rootFeature` relation identifies the `TemporalFeature` that is the root of the feature tree. Principally, it would be possible to capture the evolution if the root feature is changed. After analyzing multiple evolution histories and discussing this topic with scientists and industry experts, we decide that this

is not a common use case for which we never saw a realistic example. Thus, we deliberately decide not to capture the change of the root feature.

In contrast to standard feature models, each `TemporalFeature` can have multiple types, i.e., `TemporalFeatureTypes`. This is the case as the type of a feature may change over the course of time and, thus, at different points in time, a different type may be *temporally valid* for a feature. Thus, for a standard feature model, a feature type would be modeled as an attribute of a feature entity. However, it is not possible to define additional attributes for an attribute. Thus, we model feature types as own entity with own attributes for the temporal validity and relate it to features. The same applies for feature names (`TemporalName`) and group types (`TemporalGroupType`).

Similarly, over the course of time, a feature may have different subgroups and groups may contain different features. For standard feature models, relations between features and groups are also modeled as such in the metamodel. However, we need to attribute this relation with a temporal validity. Thus, we create own entities representing these relations but with additional information regarding their evolution. With the `TemporalParentFeatureChildGroupRelation` we determine the parent feature of a group for a specific temporal validity. For instance, if a group g_1 is located under a feature f_1 for the time span $[t_0; t_1)$ and for the interval $[t_1; t_2)$, g_1 should be located under a feature f_2, this results in two different instances of `TemporalParentFeatureChildGroupRelation`. The first one relates g_1 with f_1 and has the temporal validity $[t_0; t_1)$ and the second one related g_1 with f_2 with the temporal validity $[t_1; t_2)$. The parent feature of a `TemporalParentFeatureChildGroupRelation` is specified by adding it to the `childGroupContainers` relation of `TemporalFeature`. The child group of a `TemporalParentFeatureChildGroupRelation` is specified by setting its `childGroup` relation to that respective group. Analogously, the composition of groups, i.e., the features a group contains, is modeled using the entity `TemporalParentFeatureChildGroupRelation`.

With TFMs, we enable capturing past evolution and planning future evolution of feature models in one artifact. This is possible by setting temporal validities accordingly. If evolution is performed as part of active development, all changes entail setting temporal validities to the current date. Consequently, modeled evolution automatically becomes history in the model as time passes and the points in time of the temporal validity lie in the past. Planning feature-model evolution is possible using the same mechanism but setting the dates of temporal validities to future points in time for which the evolution is planned. Similarly, intermediate steps can be introduced performing evolution operations for points in time between the current date and planned evolution steps. This shows the flexibility of our method: with the same basic mechanisms, i.e., temporal validities, we can capture evolution history, perform active model evolution, and plan evolution steps.

Figure 3.6 shows the entire evolution timeline of the example feature model as TFM. The temporal validities are annotated with dashed lines to the relevant elements affected by evolution. Thus, each element added during evolution is annotated with a temporal validity. For instance, `goCrypt` starts to be valid starting from $t_{0.5}$ and is valid until forever, i.e., it's temporal validity is $\vartheta = [t_{0.5}; \infty)$. Similarly, for elements that are removed, the upper bound of the temporal validity is annotated. For instance, `Encfs` is deleted at $t_{0.5}$, i.e., it's temporal validity is $\vartheta = [t_0; t_{0.5})$. Similarly changed elements are annotated as well. For instance, the type of the feature `IP` is MANDATORY from the feature's introduction but is changed at t_1 to OPTIONAL. Thus, the feature `IP` has two types: one MANDATORY type with temporal validity $\vartheta = [t_{0.5}; t_1)$ and one OPTIONAL type with temporal valid-

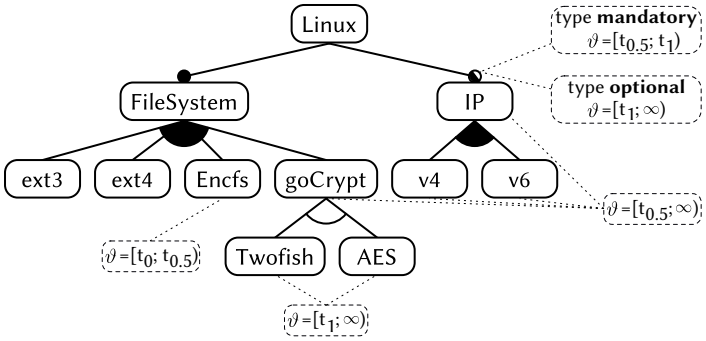


Figure 3.6.: Linux kernel feature model and its planned evolution with an intermediate evolution step as Temporal Feature Model with annotated temporal validities.

ity $\vartheta = [t_1; \infty)$. This also shows the necessity for the definition of the temporal validity as right-open interval. A feature must always have a valid type. By setting the end of a type's temporal validity to the start of another type's temporal validity, the types take turns.

3.2.2. Formalizing Temporal Feature Models

To give a precise definition of a TFM, we first give a formal definition of common feature models as a basis. Definition 3.1 defines the syntax of a feature model we use in this thesis.

Definition 3.1: Feature Model Syntax

A Feature Model is a 8-tuple $FM = (\mathcal{F}, \mathcal{G}, \lambda_\mathcal{F}, \lambda_\mathcal{G}, name, \psi, \omega, \Phi)$ with

1. $\mathcal{F}$: a finite set of features,
2. $\mathcal{G}$: a finite set of groups,
3. $\lambda_\mathcal{F} : \mathcal{F} \rightharpoonup ftype, ftype = \{optional, mandatory\}$: a function assigning a type to a feature,
4. $\lambda_\mathcal{G} : \mathcal{G} \rightharpoonup gtype, gtype = \{and, or, alternative\}$: a function assigning a type to a group,
5. $name : \mathcal{F} \rightharpoonup$ name : a function assigning a name to a feature,
6. $\psi \subseteq \mathcal{F} \times \mathcal{P}(\mathcal{G})$: relating a feature to a set of child groups,
7. $\omega \subseteq \mathcal{G} \times \mathcal{P}(\mathcal{F})$: relating a group to a set of child features (i.e., the group contains the features), and
8. Φ : a finite set of propositional formulas over $\mathcal{F}$ representing cross-tree constraints.

Each feature model consists of a set of features $\mathcal{F}$ from a universe of features $U_\mathcal{F}, \mathcal{F} \subseteq U_\mathcal{F}$. The set of groups $\mathcal{G}$ and the universe of groups $U_\mathcal{G}$ is defined analogously: $\mathcal{G} \subseteq U_\mathcal{G}$. To determine the relation of features and groups to their parent features, a feature type $ftype$ is assigned to each feature using the function $\lambda_\mathcal{F}$ and a group type $gtype$ is assigned to each group using the function $\lambda_\mathcal{G}$. For practical reasons, e.g., communication of feature models, a name is assigned to each feature using the function $name$, whereas an arbitrary character string can be assigned to a feature. We explicitly

separate the identity of features and their names as, potentially, feature names may change. The relations ψ and ω define the tree hierarchy of a feature model. ψ relates features to a set of child groups, i.e., the parent feature of a group is defined using this relation. A particularity of this notion is that we allow adding multiple child groups to a feature. Other feature-modeling notations only allow one child group per feature. ω defines the relation between groups and their child features, i.e., it defines which features are part of which group. Finally, Φ is a set of propositional formulas with features as variables that express cross-tree constraints in addition to the tree structure and the types.

In TFMs, the strict syntax of standard feature models must be relaxed to enable storing the entire evolution timeline in one model. As Figure 3.5 illustrates, most elements of a TFM have a temporal validity as attribute. Additionally, previously single-valued attributes, such as feature types, must be multi-valued for TFMs as the value can change over the course of time but we want to store the entire information. Based on the formalization of standard feature models, we define a formalization for TFMs in Definition 3.2 that relaxes the feature model syntax and adds temporal validities where necessary. A temporal validity $\vartheta = [\vartheta_{since}; \vartheta_{until})$ with $\vartheta_{since}, \vartheta_{until} \in \mathbb{R}$ is a right-open interval of real numbers whereas the values define evolution steps performed in the TFM and represent points in time. For instance, the first evolution step of a feature model is associated with 1, the second step with 2, etc. To enable the introduction of intermediate evolution steps, we use real numbers as the domain for the values. Thus, if an intermediate step is introduced between the first and second evolution step, it is associated with 0.5 as value. The set Θ is the universe of all temporal validities.

Definition 3.2: Temporal Feature Model Syntax

A Temporal Feature Model is a 12-tuple $TFM = (\mathcal{F}, \mathcal{G}, \tau_{\mathcal{F}}, \tau_{\mathcal{G}}, \lambda_{\tau_{\mathcal{F}}}, \lambda_{\tau_{\mathcal{G}}}, name_{\tau}, \psi_{\tau}, \omega_{\tau}, \Phi, \tau_{\Phi}, \Theta)$

1. $\mathcal{F}$: a finite set of features,
2. $\mathcal{G}$: a finite set of groups,
3. $\tau_{\mathcal{F}} : \mathcal{F} \rightharpoonup \Theta$: a function assigning a temporal validity to each feature,
4. $\tau_{\mathcal{G}} : \mathcal{G} \rightharpoonup \Theta$: a function assigning a temporal validity to each group,
5. $\lambda_{\tau_{\mathcal{F}}} \subseteq \mathcal{F} \times ftype \times \Theta, ftype = \{optional, mandatory\}$: relating features with a type and a temporal validity,
6. $\lambda_{\tau_{\mathcal{G}}} \subseteq \mathcal{G} \times gtype \times \Theta, gtype = \{and, or, alternative\}$: relating groups with a type and a temporal validity,
7. $name_{\tau} \subseteq \mathcal{F} \times \text{name} \times \Theta$: relating features with a name and a temporal validity,
8. $\psi_{\tau} \subseteq \mathcal{F} \times \mathcal{P}(\mathcal{G}) \times \Theta$: relating a feature to a set of subgroups and a temporal validity,
9. $\omega_{\tau} \subseteq \mathcal{G} \times \mathcal{P}(\mathcal{F}) \times \Theta$: relating a group to a set of child features (i.e., the group contains the features) and a temporal validity,
10. Φ : a finite set of propositional formulas over $\mathcal{F}$ representing cross-tree constraints, and
11. $\tau_{\Phi} : \Phi \rightharpoonup \Theta$: a function assigning a temporal validity to a cross-tree constraint.

Analogously to the feature model syntax, a TFMs consists of a set of features $\mathcal{F}$ (line 1) and a set of groups $\mathcal{G}$ (line 2) that are subsets from the respective universes $U_{\mathcal{F}}, U_{\mathcal{G}}$. As features and groups may be added or removed, a temporal validity is assigned to each feature and group using the functions $\tau_{\mathcal{F}}$ (line 3) and $\tau_{\mathcal{G}}$ (line 4). Similarly, feature and group types may change during evolution. Consequently, we must relax the notion of assigning feature types to features. The relation $\lambda_{\tau_{\mathcal{F}}}$ (line 5) relates features with types and temporal validities. Thus, a feature does not have only one type as for standard feature models but has multiple types that are valid for different time spans. The syntax for group types and feature names works analogously using the relations $\lambda_{\tau_{\mathcal{G}}}$ (line 6) and $name_{\tau}$ (line 7). If features or groups are moved in the feature tree, the parent feature of a group or the parent group of feature changes. The relations ψ_{τ} (line 8) and ω_{τ} (line 9) capture this by relating features with child groups and, respectively, groups with child features for different time spans. Finally, Φ (line 10) is the set of cross-tree constraints. To express that individual cross-tree constraints are valid for specific points in time, τ_{Φ} (line 11) is a function assigning temporal validities to cross-tree constraints. We deliberately chose the previously presented complexity of the formalization as we will need the provided expressiveness to define well-formedness properties and analyses in later chapters.

To illustrate this formalization, we formalize the TFM of the use case (cf. Figure 3.6) in Example 3.1. For simplicity, we use the feature names as identifiers of the features. For the sake of brevity, we will omit the $name_{\tau}$ relation in the example. Additionally, we omit Φ and τ_{Φ} as the use case does not contain cross-tree constraints, but this works analogously to the rest. To identify groups, we will use the parent feature of a group together with a number as part of a group's identifier and will prepend "G_".

Example 3.1: Example of Temporal Feature Model Syntax

$$
\begin{aligned}
\Theta &= \{(\vartheta_0 = [t_0; \infty)), (\vartheta_1 = [t_0; t_{0.5})), (\vartheta_2 = [t_{0.5}; t_1)), (\vartheta_3 = [t_{0.5}; \infty)), (\vartheta_4 = [t_1; \infty))\} \\
\mathcal{F} &= \{\text{Linux}, \text{FileSystem}, \text{ext3}, \text{ext4}, \text{Encfs}, \text{goCrypt}, \text{Twofish}, \text{AES}, \text{IP}, \text{v4}, \text{v6}\} \\
\mathcal{G} &= \{\text{G\_Linux}_0, \text{G\_FileSystem}_0, \text{G\_goCrypt}_0, \text{G\_IP}_0\} \\
\tau_{\mathcal{F}} &= \{\text{Linux} \rightharpoonup \vartheta_0, \text{FileSystem} \rightharpoonup \vartheta_0, \text{ext3} \rightharpoonup \vartheta_0, \text{ext4} \rightharpoonup \vartheta_0, \text{Encfs} \rightharpoonup \vartheta_1, \text{goCrypt} \rightharpoonup \vartheta_3, \\
&\quad \text{IP} \rightharpoonup \vartheta_3, \text{v4} \rightharpoonup \vartheta_3, \text{v6} \rightharpoonup \vartheta_3\} \\
\tau_{\mathcal{G}} &= \{\text{G\_Linux}_0 \rightharpoonup \vartheta_0, \text{G\_FileSystem}_0 \rightharpoonup \vartheta_0, \text{G\_IP}_0 \rightharpoonup \vartheta_3, \text{G\_goCrypt}_0 \rightharpoonup \vartheta_4\} \\
\lambda_{\tau_{\mathcal{F}}} &= \{(\text{Linux}, \textit{mandatory}, \vartheta_0), (\text{FileSystem}, \textit{mandatory}, \vartheta_0), (\text{ext3}, \textit{optional}, \vartheta_0), \\
&\quad (\text{ext4}, \textit{optional}, \vartheta_0), (\text{Encfs}, \textit{optional}, \vartheta_1), (\text{goCrypt}, \textit{optional}, \vartheta_3), (\text{v4}, \textit{optional}, \vartheta_3), \\
&\quad (\text{v6}, \textit{optional}, \vartheta_3), (\text{AES}, \textit{optional}, \vartheta_4), (\text{Twofish}, \textit{optional}, \vartheta_4), \\
&\quad (\text{IP}, \textit{mandatory}, \vartheta_2), (\text{IP}, \textit{optional}, \vartheta_4)\} \\
\lambda_{\tau_{\mathcal{F}}} &= \{(\text{G\_Linux}_0, \textit{and}, \vartheta_0), (\text{G\_FileSystem}_0, \textit{or}, \vartheta_0), (\text{G\_IP}_0, \textit{or}, \vartheta_3), \\
&\quad (\text{G\_goCrypt}_0, \textit{alternative}, \vartheta_4)\} \\
\psi_{\tau} &= \{(\text{Linux}, \{\text{G\_Linux}_0\}, \vartheta_0), (\text{FileSystem}, \{\text{G\_FileSystem}_0\}, \vartheta_0), (\text{IP}, \{\text{G\_IP}_0\}, \vartheta_3), \\
&\quad (\text{goCrypt}, \{\text{G\_goCrypt}_0\}, \vartheta_4)\} \\
\omega_{\tau} &= \{(\text{G\_Linux}_0, \{\text{FileSystem}, \text{IP}\}, \vartheta_0), \\
&\quad (\text{G\_FileSystem}_0, \{\text{ext3}, \text{ext4}, \text{Encfs}\}, \vartheta_1), (\text{G\_FileSystem}_0, \{\text{ext3}, \text{ext4}, \text{goCrypt}\}, \vartheta_3), \\
&\quad (\text{G\_IP}_0, \{\text{v4}, \text{v6}\}, \vartheta_3), (\text{G\_goCrypt}_0, \{\text{Twofish}, \text{AES}\}, \vartheta_4)\}
\end{aligned}
$$

As this example shows, we can formalize entire TFM timelines. In contrast to an individual feature-model version, this becomes complex but compared to multiple feature-model versions,

this is more concise as equivalent parts only occur once. Moreover, change can be directly seen. For instance, the different compositions of the OR group under the feature `FileSystem` can be easily seen in ω_τ as multiple triples for the group $\texttt{G\_FileSystem}_0$ with different temporal validities exist.

3.3. Evaluation

In the previous sections, we introduced Temporal Feature Models (TFMs) – a novel notation for tracking and planning feature-model evolution. We perform three different evaluations to answer **Research Question RQ1 – Modeling the Entire Feature-Model Evolution Timeline**. First, we discuss if and how TFMs fulfill the requirements we posed in Section 3.1. Second, we show the feasibility of our method by implementing it in a tool suite called DarwinSPL and by modeling the use case using DarwinSPL. Finally, we show applicability to real-world feature-model evolution modeling and importing real-world feature-model evolution as TFM.

3.3.1. Fulfillment of Requirements

In Section 3.1, we defined a set of requirements that a notation for feature-model evolution should fulfill. To show the suitability of TFMs, we will go through these requirements and elaborate if and how TFMs fulfill these requirements.

Req.1: Adding and removing features and groups. `TemporalFeatures` and `TemporalGroups` are modeled as `TemporalElement`. Thus, they can be added at a point in time t_0 by setting the beginning of their temporal validity to t_0 and they can be removed at a point in time t_1 by setting the end of the temporal validity to that point.

Req.2: Changing feature and group types. The types of `TemporalFeatures` and and `TemporalGroups` are modeled as individual classes that inherit from `TemporalElement`. Moreover, each `TemporalFeature` and `TemporalGroup` contains a set of types. Thus, if a feature type should change at point in time t, the end of the old type's temporal validity is set to t. Then, a new type is added to the set of the feature's types and the beginning of the new type's temporal validity is set to t as well.

Req.3: Changing feature names. Changing a feature name works analogously to changing a feature type as `TemporalNames` are modeled as `TemporalElement` and a `TemporalFeature` contains a set of `TemporalNames`.

Req.4: Changing the tree structure. To change the tree structure, the parent feature of a group must change, or the group membership of a feature must change. The relation between features and groups are captured by individual classes, namely `TemporalParentFeatureChildGroupRelation` and `TemporalParentGroupChildFeaturesRelation` that both inherit from `TemporalElement`. To change a parent feature from feature f_0 to feature f_1 of a group g at point in time t, the end of the temporal validity of the `TemporalParentFeatureChildGroupRelation` that relates f_0 and g must be set to t. Then, a new `TemporalParentFeatureChildGroupRelation` is created with a temporal validity that starts at t. This new relation object is added to the `childGroupContainers` of f_1 and its `childGroup` is set to g. Changing a group composition works analogously.

Req.5: Capturing past history. The past evolution history of a feature model can be captured by using temporal points for the temporal validities that lie in the past.

Req.6: Planning of future changes. Similar to capture past history, planning of future changes can be achieved by setting the temporal points of temporal validities to future points in time.

Req.7: Performing intermediate changes. The concept of temporal validities enables to perform intermediate evolution by modifying existing temporal validities and adding new elements with temporal validities that lie between the original ones. For instance, if a `TemporalFeature` f has two `TemporalNames` n_0 and n_1 with the temporal validities $\vartheta_{n_0} = [t_0; t_1)$ and $\vartheta_{n_1} = [t_1; t_2)$, respectively. To add a new name $n_{0.5}$ with $\vartheta_{n_{0.5}} = [t_0.5; t_1)$, we set $\vartheta_{n_0} = [t_0; t_{0.5})$ and add $n_{0.5}$ to the `names` of f. Intermediate changes to other aspects of a TFM work analogously.

Req.8: Capture temporal relation between evolution steps. In TFM we consider temporal points as ordered set of points in time. For instance, real dates can be used as these points in time. Thus, the temporal points of the temporal validities capture the temporal relation between the evolution steps by design.

Req.9: Direct retrieval of changes without additional computation. As we encode evolution as temporal validities in TFMs, we directly store information about changes in the model. For instance, if a feature f has the validity $\vartheta_f = [t_0; t_1)$, we know that this features has been introduced at t_0 and was removed at t_1 - without the need for additional computation.

In summary, TFMs satisfies all requirements. Thus, they are theoretically suitable for capturing and planning feature-model evolution. To show applicability, we will show the feasibility to apply this concept by providing an implementation and an application to the use case in the following.

3.3.2. Implementation

To show feasibility of capturing and planning feature-model evolution using a TFM, we implemented a toolsuite DarwinSPL[1] that uses TFMs as basis [NES17]. Using DarwinSPL, we modeled the feature-model evolution of the use case in Section 3.1. Figures 3.7 – 3.8 show screenshots of the DarwinSPL Temporal Feature Model Editor in which we modeled the use case. In the upper part of the editor, an *evolution slider* represents the feature-model timeline. To start modeling, a date has to be added to the evolution slider. Afterwards, the feature model can be created as common from other graphical feature-model editors. Figure 3.7 shows the TFM editor after modeling the initial state of the use case. To perform evolution, a new date must be added to the slider, and this date must be selected using the slider. Changes can be performed using the standard editor operations. In the backend, DarwinSPL saves the changes as changes to temporal validities of the respective TFM. Using the slider, all evolution steps of a TFM can be shown and, then, edited.

Planning feature-model evolution is possible using the same mechanisms as performing standard evolution. A new date is added but this date lies in the future. Changes are performed as previously mentioned. Figure 3.8 shows the TFM editor after adding a future date to the evolution slider and planning the future changes of the use case. However, as already outlined, intermediate changes are necessary as evolution plans must be changed, have to be refined, or are in-

[1] `https://gitlab.com/DarwinSPL/DarwinSPL`

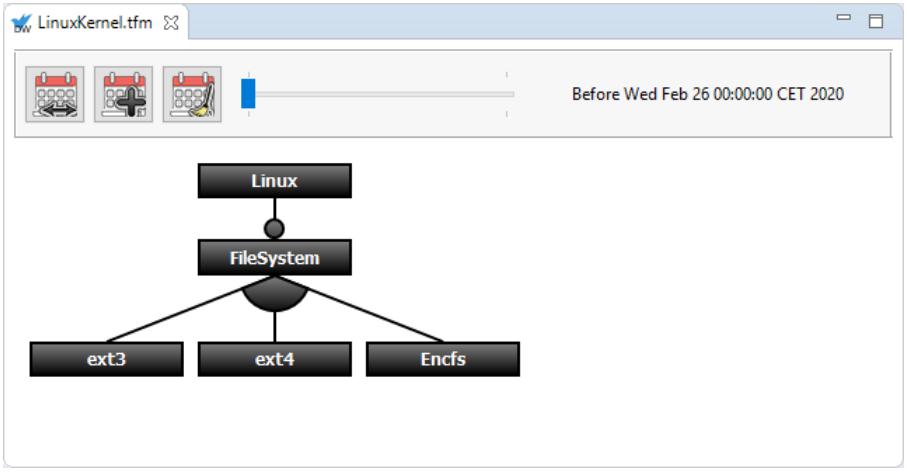


Figure 3.7.: Screenshot of the DarwinSPL Temporal Feature Model editor for the current state of the use case at t_0.

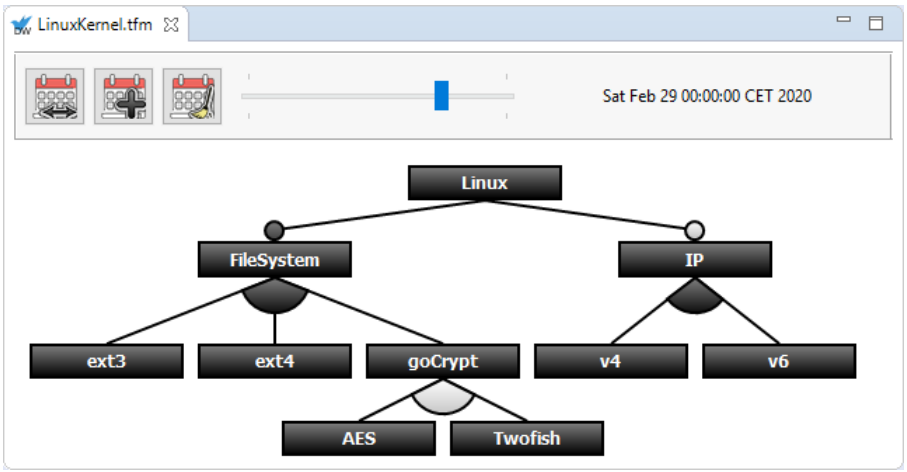


Figure 3.8.: Screenshot of the DarwinSPL Temporal Feature Model editor for the planned evolution of the use case at t_1.

crementally realized. This is where existing technologies are particularly limited. With DarwinSPL, a new date is added to the evolution slider that lies between two existing evolution steps and then, changes can be performed as engineers are used to. In the backend, DarwinSPL maintains the temporal validities, and performed changes affect the temporal validities for the date of the change. Figure 3.9 shows the TFM editor after retroactively introducing the intermediate evolution of the use case. In summary, with DarwinSPL, we provide a tool suite that enables easy modeling and planning of feature-model evolution which enabled us to capture the use case of Section 3.1. To the best of our knowledge, no other feature-model editor exists that is capable of illustrating the structural feature-model evolution.

The heterogeneity of different notations to capture feature-model evolution impede reuse, collaboration, and common analysis methods that are highly performant. This problem has also been identified by Marques et al. [MSR+19] in a literature survey on SPL evolution. The authors argue that "the SPL community needs to work together to improve the state of the art, creating methods and

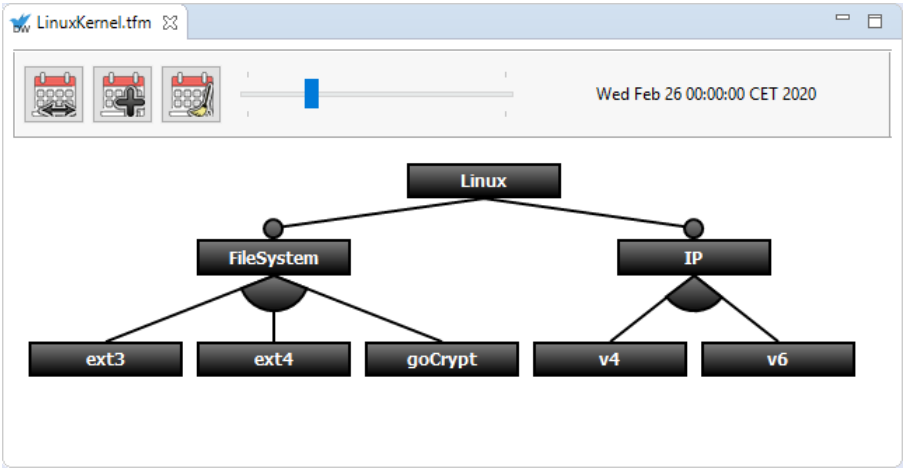


Figure 3.9.: Screenshot of the DarwinSPL Temporal Feature Model editor for the intermediate evolution step of the use case at $t_{0.5}$.

tools that support SPL evolution more comparably" [MSR+19]. To address this barrier, we implemented a harmonization framework for multiple feature-model evolution notations [HNL+19]. This framework provides a generic API that provides methods to perform evolution operations on feature models. To utilize a feature-model evolution notation with this API, several methods must be implemented that delegate operation calls to notation-specific methods. Moreover, we implemented an operation model that persists each evolution operation to replay it using any other language.

Based on this API, we devised a feature-model editor and simple analyses. In the end, we implemented the API for Hyper Feature Models [SSA14b], FORCE [HPL+18], and TFMs. Additionally, the API enables to integrate feature-model notations that are unaware of evolution, such as FeatureIDE [MTS+17] or pure:variants feature models [Gmb06]. In summary, the feature-model editor worked for all these notations and we were able to replay operations that we originally performed on one notation to create the same feature model for another notation. However, not all operations were provided by each notation. The only notation that supported all operations were TFMs which also shows their expressiveness and flexibility.

3.3.3. Applicability to Real-World Feature-Model Evolution

To verify whether TFMs are applicable for real-world feature models and their evolution, we model this evolution using TFMs. Optimally, we would reproduce this evolution directly using a TFM and DarwinSPL. However, this requires domain knowledge as some evolution operations are ambiguous and, thus, engineers would have to use TFMs from the beginning. For instance, without domain knowledge, the renaming of a feature cannot be distinguished from removing a feature with the old name and adding a feature with the new name. As we do not have the opportunity to let real-world developers use TFMs for modeling the evolution of their feature models, we import multiple version snapshots of real-world feature models as one sole TFM.

Table 3.1.: Numbers of features and groups of two real-world feature model versions.

		V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	Aggregated	TFM
FinancialServices01	# Features	557	704	712	711	716	712	759	771	774	771	7,187	1,083
	# Groups	124	153	155	155	160	162	179	183	183	184	1,638	259
Automotive02	# Features	14,010	17,742	18,434	18,616	-	-	-	-	-	-	68,802	24,053
	# Groups	1,395	1,703	1,774	1,788	-	-	-	-	-	-	6,660	1,839

In particular, we import the histories of two feature models: one of a financial company[2] (FinancialServices01) and one of a company from the automotive domain[3] (Automotive02). Table 3.1 shows the numbers of features and groups of the individual feature model versions. The financial company's feature model comprises ten versions and the automotive feature model four versions. We imported the different versions of each feature model into one single TFM by incrementally applying comparing methods. To this end, we match features in two versions by comparing their name. As no other information exists about the feature identity, we lose precision as we cannot detect a feature renaming. However, this is an explicit drawback of using version instead of a TFM. If we cannot find a matching partner for a feature in the older version, we assume that this feature has been deleted in the new version and set the end of its temporal validity in the TFM accordingly. Vice versa, if we do not find a matching partner for a feature in the new version, we assume that this feature is newly introduced in that version. In the next step, we investigate whether feature and group types have changed. Afterward, we check whether features have been moved to other groups. As in the data sources, no group identities exist, we cannot identify whether a group has been moved or whether all of its features have been moved. This is again a drawback of not using TFMs. Thus, we only consider moving groups. We successfully imported all versions for each feature model in a single TFM respectively. To validate correct modeling, we exported each feature-model version from the TFM and compared it to the original versions.

One benefit of TFMs is that differencing becomes obsolete as all changes are directly captured as temporal elements. For differencing methods, all versions must be compared with each other, i.e., matches between all features and groups must be found. Consequently, many feature model elements must be compared which is not necessary with TFMs. To give an estimate of this benefit, we compare the aggregated number of the features and groups of all individual feature-model versions with the number of features and groups of the TFM (cf. Table 3.1). Naturally, optimized differencing mechanisms do not have to compare each possible pair of features as they can exploit the feature model tree structure and other heuristics. Nonetheless, a naive approach would compare all possible candidates with each other. The numbers indicate that TFMs can significantly reduce the potentially necessary overhead to compare multiple elements. In summary, we can confirm that TFMs can correctly capture real-world feature-model evolution.

[2] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples/FeatureModels/FinancialServices01

[3] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples/FeatureModels/Automotive02_V1 (three more versions in the same repository)

3.3.4. Threats to Validity

Our evaluation is subject to internal and external threats to validity. A threat to internal validity is that we could not evaluate the suitability for the planning of real-world evolution. This is because, to the best of our knowledge, no data exists which would enable us to evaluate the planning. However, after discussing the planning and evolution activities of our industry partners, we gained experience in how real-world evolution is planned and performed. We used this experience to design the use case in Section 3.1.

The external validity is affected as we only used two large-scale real-world feature models with their evolution in our evaluation. Thus, the representativity of the used data may be limited. However, we are not aware of other real-world feature models with evolution. As the used data stems from very different domains (i.e., automotive and financial sector) and with different modeling styles, we expect that our results are applicable for most real-world feature models.

3.4. Related Work

State of Practice Typically, the history of feature models is stored using Version Control Systems (VCSs). Reasoning on such histories is possible by retrieving multiple model versions and performing a difference comparison between them [KKO+12, BKL+16]. This is very important as it is very hard for engineers to understand evolution if they do not exactly know how the feature model changed [BKL+16]. However, differencing is only an approximation as some changes cannot be exactly determined. For instance, if a feature has been renamed, differencing mechanisms cannot know whether this has been a renaming or whether the old feature has been removed and a new feature has been added. Moreover, the main necessity to analyze these differences is that many changes to the feature model are performed in an ad-hoc many and, thus, are not documented [BKL+16]. Future planning with VCSs is only possible via workarounds, e.g., by adding a new version branch that is integrated into the main branch at some point. However, this makes it hard to incrementally realize the planned evolution as the entire branch is merged. Moreover, introducing intermediate evolution steps is problematic as the chronological relation between changes might not be clear as VCSs do not know the concept of time.

Product-Line Evolution Visions Passos et al. [PCA+13] present a vision to evolve product lines based on feature-model evolution. They argue that feature models are highly suitable as a starting point for the evolution of product lines. Such an approach would enable us to trace the evolution of other artifacts, analyze evolution, and provide recommendations on how to evolve other product line artifacts such as configurations. The abstraction level of feature models enables many stakeholders to understand the product line evolution and it makes feature models a central artifact to discuss evolution.

Feature-Model Evolution Analyses Thüm et al. [TBK09] categorize feature-model changes based on their impact of possible valid configurations. They define three categories: *refactorings* if no valid configurations are added or removed, *generlizations* if new valid configurations are added, *specialization* if originally valid configurations are removed, and *arbitrary edits* if some new valid configurations are added and some originally valid configurations are removed. These categories can be used to improve analyses of feature models in presence of evolution as for some analyses, no new computations are necessary if a change falls in a certain category. However, they

do not model feature-model evolution nor evolution operations but retroactively analyze differences between two feature-model snapshots. If evolution would be modeled explicitly, these computations could be performed more efficiently.

Gamez and Fuentes [GF11] analyze the impact of feature-model evolution operations in terms of necessary changes to existing configurations. In particular, they identified several relevant evolution scenarios: adding and removing features, adding and removing features to/from groups, modifying feature types, and adding and removing cross-tree constraints. They claim that modifications to features, such as modifying their types or moving features in the tree, can be expressed by adding and deleting features. While this is true on a syntactical level, on a semantical level, these are very different operations which are relevant for certain analyses. Gamez and Fuentes [GF11] represent feature-model changes as changes to resulting constraints. In the end, they calculate how existing configurations must be changed in terms of dropped or added constraints and use this as a measure for the effort to update products. This method is very limited as it only considers existing configurations, and analyzing the impact on all configurations is not possible as enumerating all possible configurations is infeasible due to combinatorial explosion [MR14a]. Moreover, changed features in configurations are not necessarily a suitable measure for the effort to update products as it differs from feature to feature how expensive it is to add or remove them.

White et al. [WGS+14] identified the necessity for long-term evolution planning of configurable systems. In particular, they consider how configurations and configuration processes change due to feature-model evolution. To this end, they provide a method to encode feature-model evolution as Constraints Satisfaction Problem (CSP). Similar to Gamez and Fuentes [GF11], changes between feature-model versions are represented by added or removed constraints in the CSP by encoding the constraints with identifiers for evolution steps. However, they do not provide any modeling language to capture the entire feature-model timeline. Thus, this is rather a theoretical backend that can be used for analysis purposes. In the chapters of this thesis dealing with analyses, we will come back to the work of White et al. [WGS+14].

Retroactive Modeling of Feature-Model Evolution Bürdek et al. [BKL+16] retroactively derive performed evolution operations by utilizing a differencing mechanism between two feature-model versions. They argue that it is important to know which operations have been performed for engineers to understand evolution. Moreover, knowing the evolution operations enables to reason about the feature-model evolution. The main motivation of Bürdek et al. [BKL+16] is that many changes to feature models are performed in an ad-hoc manner and, thus, are not documented. However, proper documentation is necessary to understand and reason about the evolution. Additionally, the authors provide a comprehensive catalog of typical evolution operations for feature models. To derive evolution operations, they compute the differences between two feature-model versions and map them to the evolution operations. In contrast to the previously mentioned research, Bürdek et al. [BKL+16] model concrete evolution operations. However, the operations are retrieved retroactively and as for all differencing mechanisms, this is only an approximation as for some differences, multiple evolution operations are theoretically possible. Moreover, retroactive operation retrieval does not support planning activities.

Automatic Tracking of Feature-Model Evolution History With Feature-Driven Versioning, Mitschke and Eichberg [ME08] provide a method to capture feature-model evolution. To this end, two

types of versions are used: feature versions and implementation artifact versions. Thus, if a feature is changed, e.g., by changing its type, its version number is increased. Similarly, if implementation artifacts mapped to a feature are changed, the respective implementation artifact version of that feature is increased. However, in both cases, no information on how a feature or an implementation artifact changes is stored or provided. Similarly, Schwägerl and Westfechtel. [SW16] introduced SuperMod that uses version control mechanisms as backend and automatically keeps track of the evolution.

Proactive Modeling of Feature-Model Evolution Seidl et al. [SSA14b, SSA14c] explicitly model product-line evolution with Hyper Feature Models (HFMs). In particular, they introduce feature versions that represent different implementations of a specific feature. Thus, the different versions are mapped to different realization artifacts. However, they do not cover the evolution of the feature model itself. Additionally, they analyzed the co-evolution of models and feature mappings [SHA12]. To this end, they defined multiple co-evolution operations that are also applicable to feature models. For instance, the *remove feature mapping* operation is similar to the *remove feature* operation.

Hinterreiter et al. [HPL+18] introduced the feature-modeling notation FORCE that is capable of capturing feature-model evolution in one model. In FORCE, changes to the feature model are captured as feature-model versions. Each of these versions represents an evolutionary step of the feature model. The evolution itself is captured similar to version control systems and, thus, as snapshots. The technological similarity to version control systems is a strong limitation of this method. Differences between feature-model versions must be explicitly computed by comparing two versions. Additionally, no planning is explicitly supported and, thus, no means to introduce intermediate evolution steps exist.

With EvoFMs, Botterweck et al. [BPP+09, BPD+10, BP14], Schubanz et al. [SPB+12, SPP+13], and Pleuss et al. [PBD+12] introduced a method to capture the evolution of feature models and put a focus on evolution planning. EvoFMs are models that represent the evolution of a feature model. An EvoFM is a feature model itself and its features represent fragments that evolve in the original feature model. These fragments can be entire subtrees consisting of multiple features. The types of the EvoFM features indicate whether these fragments are available for the entire feature-model timeline, i.e., MANDATORY type, or only for certain time spans, i.e., OPTIONAL type. In addition to the EvoFM, an evolution plan indicates the evolution steps of the feature model. The evolution plan is a Gantt-style diagram with the evolution steps as the x-axis. The y-axis consists of EvoFM features. The bars indicate whether the respective fragments in the original feature model exist for the covered evolution steps. Apart from EvoFM features, the evolution plan may also contain operations on the original feature model, such as move features, or operations on cross-tree constraints. Pleuss et al. [PBD+12] extended EvoFMs with evolution rationales that explain why evolution operations were performed.

EvoFMs have several shortcomings. For one, the concept of fragments that the EvoFM features represent is very inflexible. With this concept, the evolution of all elements covered by fragments can only be performed together. Thus, if single elements of the fragment should evolve differently than others, the fragment has to be split up. This requires adaptation of the entire EvoFM and the evolution plan. Moreover, operations on the feature-model level, such as move feature, are modeled explicitly. Consequently, exactly these operations must be supported. This also makes analyses complicated as all operations modeled before for points in time before the version to ana-

lyze must be applied. Finally, it is very complex to have three different models: the original feature model, the EvoFM, and the evolution plan. This makes it very hard for engineers to understand evolution. However, the visualizing feature-model evolution as a Gantt-diagram can be very helpful for engineers to comprehend and plan evolution.

Comparison of Feature-Model Evolution Languages In the following, we compare the existing notations with TFMs by highlighting which requirements of Section 3.1 are addressed by which notation. In particular, we compare TFMs with Hyper Feature Models by Seidl et al. [SSA14b], Feature-Driven Versioning by Mitschke and Eichberg [ME08], with FORCE by Hinterreiter et al. [HPL+18], with SuperMod by Schwägerl and Westfechtel [SW16], and with EvoFM by Botterweck et al. [BPP+09, BPD+10].

Table 3.2.: Comparison of feature-model evolution notations in respect to the requirements of Section 3.1.

	Hyper Feature Models	Feature-Driven Versioning	SuperMod	FORCE	EvoFM	TFMs
Req.1: Adding and removing features and groups	-	+	+	+[b]	+	+
Req.2: Changing feature and group types	-	+	+	+[b]	+	+
Req.3: Changing feature names	-	+	+	-	+	+
Req.4: Moving features and groups	-	+	+	+[b]	+	+
Req.5: Capturing history	+	o[a]	+	+	+	+
Req.6: Planning future changes	-	-	-	-	+	+
Req.7: Intermediate changes	-	-	-	-	o[c]	+
Req.8: Temporal Relation between evolution steps	+	-	-	-	+	+
Req.9: Direct retrieval of changes without additional computations	+	-	-	-	-	+

[a]: only increases feature version numbers but does not save history.
[b]: does not have an explicit concept of groups.
[c]: not considered explicitly and only possible with significant effort.

Table 3.2 shows an overview of the different modeling notations and how they fulfill the requirements of Section 3.1. Most notations enable to add and remove features and groups (**Req.1**), to change feature and group types (**Req.2**), to change feature names (**Req.3**), and to move features and groups (**Req.4**). As Hyper Feature Models only support feature versions and no changes to the feature-model structure, these requirements are not fulfilled. Moreover, FORCE has several limitations. For one, they do not have a distinct concept of groups but groups are implicitly identified by child features of the same type under a parent feature. Thus, it is possible to have a maximum of one child group per type under each feature, e.g., one OR group. Moreover, FORCE does not support the renaming of features.

All notations can store the history of evolution (**Req.5**). For Hyper Feature Models, this only includes feature versions. For Feature-Driven Versioning, this capability is extremely limited, as only

version counters are increased but more information on the change is not stored. Thus, it is only possible to know *that* something changed but not *how* it changed.

The only notation except for TFMs that supports future evolution planning are EvoFMs (**Req.6**). It is also explicitly supported to plan changes for future dates. In theory, EvoFMs also support the introduction of intermediate steps as only notation except for TFMs (Req.7). However, this is not explicitly supported and, thus, only possible with workarounds. Consequently, multiple models must be adapted and inconsistencies can be easily introduced.

The temporal relation between evolution steps (**Req.8**) cannot be explicitly captured by most notations. Only the feature versions of Hyper Feature Models, EvoFMs, and TFMs support this functionality. In Hyper Feature Models, the relation between feature versions can be captured in a version tree. For EvoFMs, real dates are used and, thus, the temporal relation is defined by design. For TFMs, abstract versions with an order as well as real dates can be used for temporal validities.

Finally, the direct retrieval of changes without the need to perform additional computations (**Req.9**), such as differencing, is only supported by Hyper Feature Models and TFMs. For the other notations, changes must be computed by performing a differencing analysis between multiple versions or by applying a list of evolution operations. For Feature-Driven Versioning, even this approach is not possible as this notation does not store changes at all but only automatically increases version numbers.

With TFMs, we can capture all structural changes as we modeled each part of a feature model as *temporal element*. The concept of temporal validities for temporal elements allows capturing both history and planned changes using the same constructs. To plan future changes, the temporal validity of changed elements is set to future points in time. Moreover, a TFM is a living artifact. As time passes, planned changes become present and, afterward, they become history. Finally, as changes are directly captured as temporal validities of model elements, the changes can be precisely retrieved without additional computational effort, such as the application of operations. However, for some analyses and scenarios, information on performed operations are relevant. With TFMs operations can be modeled as well with their impact on temporal validities. However, as the effect is captured as changes to temporal validities, the evolution of TFMs is independent of concrete operations and, thus, TFMs are compatible with different notions of operations.

Our literature study is also subject to internal validity as we may have missed existing notations to model feature-model evolution. Thus, notations may exist that already cover the requirements of Section 3.1. To mitigate this threat, we searched for publications in relevant conference proceedings and journals that deal with variability, SPLs, and modeling. Moreover, we used snowballing based on the notations we found to find further publications. To the best of our knowledge, no further feature-model evolution notations exist. Even if that would be the case, TFMs would still fulfill the requirements and we were able to capture real-world feature-model evolution.

3.5. Chapter Summary

With TFMs we provide powerful means to capture entire feature-model evolution timelines. Consequently, we can answer **RQ1 – Modeling the Entire Feature-Model Evolution Timeline** with TFMs which also meet **Challenge 1: Modeling Feature-Model Evolution Timelines**. A particular focus of TFMs is the planning of evolution while still being able to perform changes to the current state. Moreover, TFMs are living artifacts. Plans can be changed or refined, and future evolution auto-

matically becomes present and afterward past. We provided a metamodel and a formalism that are a precise description of TFMs and allow to implement it for different use cases. Additionally, with DarwinSPL, we provide a tool suite that enables easy modeling of Temporal Feature Model (TFM) without the need to understand the complexity of the underlying notation. In our evaluation, we have shown that TFMs are superior to existing feature-model evolution notations concerning the challenges and requirements that we identified. Finally, we imported existing real-world feature-model histories as single TFMs and, thus, we have shown general applicability of our method.

Even if TFMs are superior to other feature-modeling languages for evolution, existing notations may be widespread and used in many projects. Thus, we want to provide mechanisms that transport the concept of temporal elements to other notations as well. As we strive for generalization, we want to make this concept applicable to arbitrary modeling languages and, thus, be able to support sophisticated evolution modeling for other artifacts than feature models as well. We will address this challenge in the following chapter.

As TFMs enable planning of feature-model evolution, replanning becomes inevitable. Thus, respective methods are required and, when replanning, intermediate operations are introduced that pose the risk of introducing inconsistencies in already planned future evolution steps. Moreover, feature modeling in general poses the risk to introduce design flaws, called *anomalies* [BSR10]. With the process of evolution and growing feature models, the risk of introducing such anomalies increases. In Part III, we will provide analyses methods that ensure consistency, and detect and explain anomalies for entire feature-model evolution timelines.

On the basis of TFMs, several future research directions are possible. For some use cases, features are only temporarily suspended and will be reactivated later. For instance, if a company removes a feature but customers require it to be reintegrated. This is fundamentally different from removing and adding features with the same name. Thus, we consider to provide a more flexible notion of temporal elements. Each temporal element may have a disjunct set of temporal validities. This requires also tools, such as DarwinSPL to be adapted to be able to reactive features.

Another possible research direction is collaborative modeling of TFMs. This requires to define multiple roles or identities of contributors, and modifications are associated with a certain role or identity. Additionally, concepts for merging multiple changes that occurred in parallel are required. This is a similar challenge as general TFM development in branches. Following common development practices, TFMs can be used to model multiple possible evolution timelines.

Part III.

Analyzing Feature-Model Evolution Timelines

4 Paradox-Free Feature-Model Evolution Planning

The contents of this chapter are largely based on the work published in [NST18, HNS+20].

Summary *SPL evolution is a large endeavor and, thus, this evolution is planned far ahead. Plans need to be adapted as not all details have been known in advance or as requirements change. Consequently, it is pivotal to be able to replan feature-model evolution. Conceptually, this is possible using TFMs. However, when replanning, intermediate evolution steps are introduced that change the basis for evolution steps planned for subsequent points in time. This can lead to structural inconsistencies, denoted as evolution paradoxes, which lead to high fixing costs or even make other already planned evolution steps useless. Avoiding evolution paradoxes it thus paramount when replanning. In this chapter, we contribute a method for replanning feature-model evolution while ensuring consistency of all planned evolution steps. We provide formal specifications of TFM well-formedness rules and a formal specification of evolution operations in an execution semantics that detects evolution paradoxes before being introduced. This method is integrated into our tool suite* DarwinSPL *which enables replanning while preventing the introduction of evolution paradoxes. We assessed our method empirically and evaluated its scalability using existing feature-model evolution scenarios.*

An SPL is a particularly long-living software system as the initial development is expensive which pays off if many products are implemented and if it is used over a long period of time [BPD+10, BP14, BCM+04, FK05]. For a company that develops an SPL, this SPL is a major strategic asset and, thus, its evolution needs to be thoroughly planned. With TFMs, we presented a method to plan feature-model evolution in Chapter 3. For long-term feature-model evolution plans, it is inevitable to adapt or extend these plans and to incorporate short-term changes [WGS+14, BP14]. When planning feature-model evolution long in advance, not all details are clear. Consequently, these plans must be refined when more details are known. The plans have to be changed if these details require adaptation. Moreover, short-term requirement changes must be integrated as well. For instance, implementation obstacles lead to changed or delayed functionality, or short-term management decisions or new legal restrictions require quick realization. Figure 4.1 shows the planned evolution of the Linux kernel feature model running example. At time point t_0, an initial version is defined as depicted by arrow ①. In a second step, a plan is devised for the point in time t_1 which provides to add two alternative encryption algorithms for the `Encfs` file system. This planning step is depicted by arrow ②. For time point $t_{0.5}$ (depicted by arrow ③), the evolution is replanned by an intermediate evolution step. In this intermediate step, `Encfs` is replaced by the more recent encryption file system `goCrypt`. As a result, `Encfs` is deleted and `goCrypt` is added. As this example illustrates, replanning of feature-model evolution is pivotal to enable long-term planning while accommodating new requirements or technical developments.

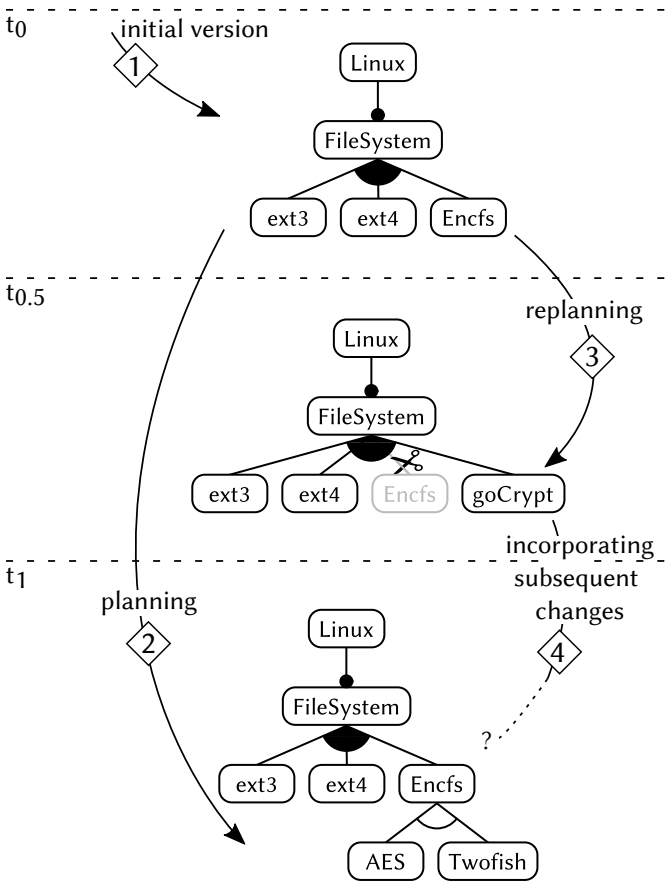


Figure 4.1.: Definition of a feature model and an evolution plan for file system features of the Linux kernel. A retroactively introduced intermediate step violates consistency.

Replanning of feature-model evolution may lead to severe inconsistencies. If an evolution step of a feature model is changed, the basis of all the following evolution steps is changed as well. For instance, Figure 4.1 shows that the feature `Encfs` is deleted in an intermediate step (arrow ⟨3⟩) while in the original plan (arrow ⟨2⟩) two new sub features are added as children of `Encfs`. Thus, the plan for time point t_1 bases on the assumption that `Encfs` exists. However, in the intermediate step, `Encfs` is removed at $t_{0.5}$ and, consequently, the basis for the original plan scheduled for t_1 has changed. Thus, retroactively changed evolution steps or newly added intermediate evolution steps lead to changed bases for future evolution steps.

However, a changed basis may lead to inconsistencies in already planned evolution steps. For instance, in Figure 4.1, when incorporating the changes of the intermediate step with the original plan (depicted by arrow ⟨4⟩), the two newly added feature of the original plan, `AES` and `Twofish`, would have no parent feature as `Encfs` is deleted in the intermediate step. Without additional knowledge, it is unclear whether the two newly added encryption algorithms `AES` and `Twofish` should become children of `goCrypt`, whether they should not be introduced at t_1, or something different. We denote such inconsistencies as *evolution paradoxes* [NST18] [HNS+20]. An evolution paradox may lead to severe costs as already planned evolution has to be revised which may involve many changes. Even worse, detecting an evolution paradox is a challenging task as the changes of an intermediate step may lead to an evolution paradox far in the future. For large feature models

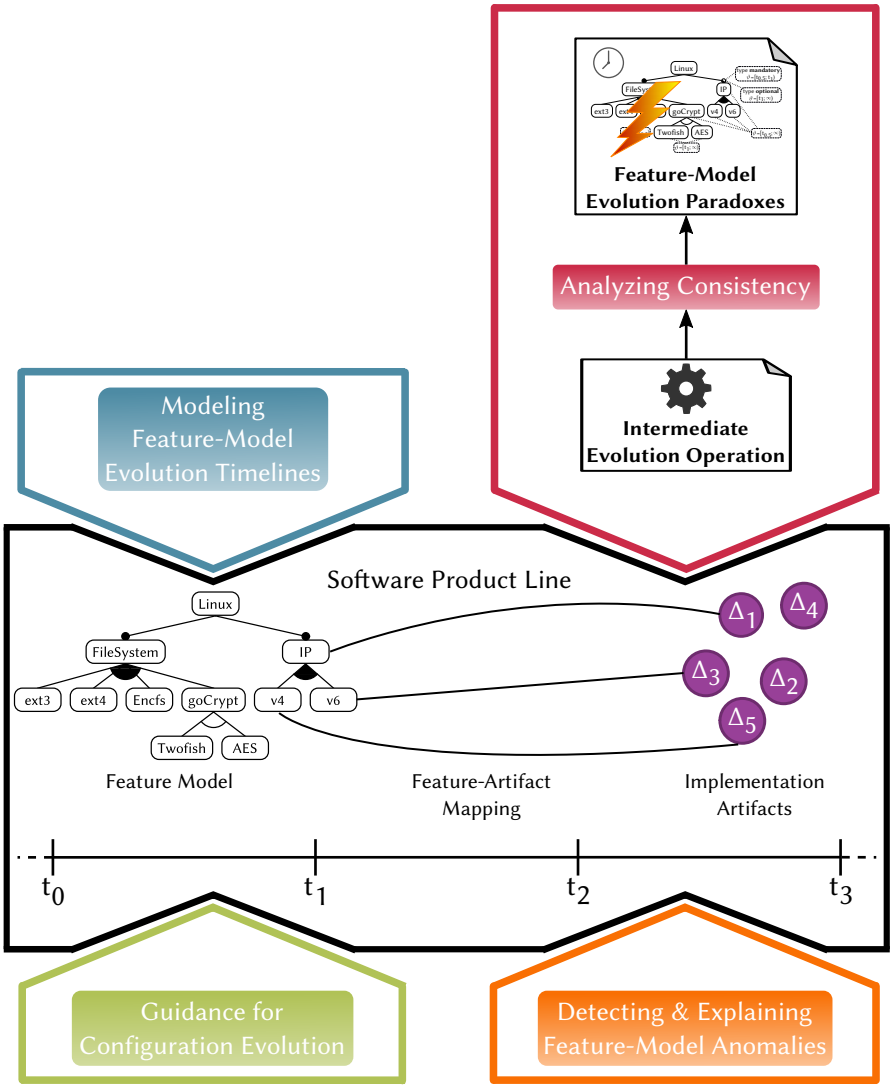


Figure 4.2.: Contribution overview – Step 2, Analyzing Evolution Consistency.

with many evolution steps, it is hard to detect such a paradox at all. Thus, automatically detecting evolution paradoxes is pivotal to enable replanning of feature-model evolution which is covered by **Challenge 2: Consistent Feature-Model Evolution Replanning**.

In this chapter, we provide a method to enable the definition of paradox-free evolution plans for feature models which answers the first part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**. Figure 4.2 shows the contributions of this thesis with more details on analyzing evolution consistency. After modeling the entire feature-model evolution timeline using a TFM, this evolution potentially needs to be replanned. The next step is to ensure consistency of the devised TFM evolution steps their replanning. In particular, we guarantee up-front whether applying an evolution operation would lead to an inconsistency. Thus, if an intermediate operation is introduced, this operation and all planned following operations are analyzed and if an evolution paradox is detected, the execution of the newly introduced operation is prohibited. As evolution paradoxes are structural inconsistencies that are introduced by intermediate evolution operations in evolution plans, we formalize TFM well-formedness rules and evolution plans in terms

of evolution operations in Section 4.1. To detect evolution paradoxes, we determine in Section 4.2 which types of paradoxes exist and which evolution operations may cause them. Finally, we define paradox-free execution semantics of the evolution operations in Section 4.3. We show feasibility of our concepts by providing an implementation integrated into the TFM modeling tool suite DarwinSPL in Section 4.4.1 and evaluate its scalability by applying our implemented method to multiple existing feature-model version histories in Section 4.4.2. Moreover, we evaluate our method by empirically showing the importance of detecting evolution paradoxes using interviews with industrial partners and an online questionnaire in Section 4.4.3.

4.1. Well-Formedness of Temporal Feature Models

In state-of-the-art feature modeling languages and tools, evolution is performed by creating a new version and modifying this new version. Preserving the structural consistency of the feature model is simple if only the current state needs to be considered. To this end, each modification operation can be defined to maintain a set of well-formedness rules. In contrast to development with standard features models, TFMs provide significant more expressive power regarding evolution. In particular, the entire evolution of a feature model is stored in a TFM by using temporal validities. Thus, after applying an evolution operation on a TFMs, the structural consistency must be ensured for each point in time. To this end, we lift the well-formedness rules for standard feature models (cf. Chapter 2) to express these properties in the presence of evolution, i.e., for each relevant point in time. Additionally, some consistency properties for feature models automatically hold by definition of the feature model formalization in Definition 3.1 (cf. Section 3.2.2). In particular, each feature and group has exactly one type which is ensured by the function assigning a type to each feature/group. The same applies to feature names. However, for TFMs, this does not hold by definition as a feature or group may have different types over the course of time. Consequently, in Definition 3.2, we related features and groups to sets of types and temporal validities instead of a one-to-one function.

We define a set of auxiliary functions that enable us to define more concise well-formedness rules for TFMs. For simplicity, we assume for these functions that only one feature type, one group type, and one feature name are temporally valid at a given point in time. We ensure these properties with the first three well-formedness rules and use them in the auxiliary functions for later rules. We define the following functions:

- $name_\tau(f,t)$ retrieves the temporally valid name of a feature f at the point in time t.
- $\lambda_{\tau_\mathcal{F}}(f,t)$ retrieves the temporally valid type of a feature f at the point in time t.
- $\lambda_{\tau_\mathcal{G}}(g,t)$ retrieves the temporally valid type of a group g at the point in time t.
- $\omega_\tau(g,t)$ retrieves the temporally valid set of child features of a group g at the point in time t. Theoretically, ω_τ may relate multiple child feature sets that are temporally valid at a given point in time to a group. Then, this function produces the union of these sets.

In Definition 4.1, we define the well-formedness rules for TFMs and, in the following, we elaborate on these rules with a particular focus on the differences to the standard feature model rules.

$WF_\tau 1$ A feature must have a type. With TFMs, multiple types that are valid at different points in time can be assigned to a feature. A TFM is consistent only if a feature has exactly one tem-

porally valid type at each point in time when the feature itself is temporally valid. In the well-formedness rules, we define that for each feature, the number of related feature types must be one for each individual point in time of the feature's temporal validity.

WF*$_\tau$*2 Analogously to features, a group must have exactly one type at each point in time of the group's temporal validity.

WF*$_\tau$*3 Feature names may change to express changed functionality or for other reasons, e.g., for marketing purposes. To ensure consistency, each feature may only have one temporally valid name which is formalized analogously to the feature types.

WF*$_\tau$*4 As the root feature of a feature model must be MANDATORY, the root feature of a TFM must be MANDATORY at each point in time when the root feature is temporally valid. To this end, we utilize one of the previously defined auxiliary functions to retrieve the type of the root feature at each point of its validity and define that it must be MANDATORY.

WF*$_\tau$*5 Feature names in a feature model must be unique. The same property must hold for TFMs at each point in time, whereas only temporally valid features are considered. Thus, we define that for each point in time of each existing temporal validity, pairwise different temporally valid features must have different names that are valid at that time.

WF*$_\tau$*6 To ensure that a TFM is a tree at each given point in time, each feature, except for the root, must be part of exactly one temporally valid group at each point in time the feature is temporally valid. To this end, we define that for each feature and each point in time of its temporal validity, the feature is either the root or it is part of exactly one relation from a group to its sub-features. Additionally, a group of that relation must be temporally valid as well as it is an inconsistency if a temporally valid feature is a child of a temporally invalid group.

WF*$_\tau$*7 Another rule to ensure tree properties of a TFM is that for each point in time a group is temporally valid, the group must have exactly one parent feature that is temporally valid. We formalize this rule analogously to *WF*$_\tau$*6*.

WF*$_\tau$*8 To ensure that each group with type ALTERNATIVE or OR contains at least two features, we consider each point in time when that group is valid and retrieve its type for that time using an auxiliary function. Then we retrieve the set of sub-features at that point in time using another auxiliary function and define that the cardinality of that set must be equal or greater than two.

WF*$_\tau$*9 As no ALTERNATIVE or OR group must contain a MANDATORY feature, we must consider each point in time in which a group has the mentioned type analogously to *WF*$_\tau$*8*. Then, we consider each temporally valid sub-feature using the auxiliary function $\omega_\tau(g, t)$ and define that their type at the considered point in time must be OPTIONAL.

Definition 4.1: Temporal Feature Model Well-Formedness

Let $f_R \in \mathcal{F}$ be the root feature of a temporal feature model $TFM = (\mathcal{F}, \mathcal{G}, \tau_\mathcal{F}, \tau_\mathcal{G}, \lambda_{\tau_\mathcal{F}}, \lambda_{\tau_\mathcal{G}}, name, \psi_\tau, \omega_\tau, \Phi, \tau_\Phi, \Theta)$. A TFM is well-formed iff:

$WF_\tau 1$ Each feature must have one type at each time point the feature is temporally valid:
$\forall f \in \mathcal{F} : \forall t \in \tau_\mathcal{F}(f) : |\{(f, type, \vartheta) \in \lambda_{\tau_\mathcal{F}} \mid t \in \vartheta\}| = 1$

$WF_\tau 2$ Each group must have one type at each time point the group is temporally valid:
$\forall g \in \mathcal{G} : \forall t \in \tau_\mathcal{G}(g) : |\{(g, type, \vartheta) \in \lambda_{\tau_\mathcal{G}} \mid t \in \vartheta\}| = 1$

$WF_\tau 3$ Each feature must have one name at each time point the feature is temporally valid:
$\forall f \in \mathcal{F} : \forall t \in \tau_\mathcal{F}(f) : |\{(f, \text{name}, \vartheta) \in name_\tau \mid t \in \vartheta, \vartheta \in \Theta\}| = 1$

$WF_\tau 4$ The root feature must be always MANDATORY:
$\forall t \in \tau_\mathcal{F}(f_R) : \lambda_{\tau_\mathcal{F}}(f_R, t) = mandatory$

$WF_\tau 5$ The names of all temporally valid features at a time point must be different:
$\forall \vartheta \in \Theta : \forall t \in \vartheta : \forall f_1, f_2 \in \mathcal{F} : (f_1 \neq f_2 \wedge t \in \tau_\mathcal{F}(f_1) \wedge t \in \tau_\mathcal{F}(f_2)) \implies name_\tau(f_1, t) \neq name_\tau(f_2, t)$

$WF_\tau 6$ For each point in time a feature is temporally valid, except the root feature, it has exactly one temporally valid parent group:
$\forall f \in \mathcal{F} : \forall t \in \tau_\mathcal{F}(f) : f = f_R \vee |\{(g, F_{\omega_\tau}, \vartheta) \in \omega_\tau \mid f \in F_{\omega_\tau} \wedge t \in \vartheta \wedge t \in \tau_\mathcal{G}(g) \wedge g \in \mathcal{G}\}| = 1$

$WF_\tau 7$ For each point in time a group is temporally valid, the group has exactly one temporally valid parent feature:
$\forall g \in \mathcal{G} : \forall t \in \tau_\mathcal{G}(g) : |\{(f, G_{\psi_\tau}, \vartheta) \in \psi_\tau \mid g \in G_{\psi_\tau} \wedge t \in \vartheta \wedge t \in \tau_\mathcal{F}(f) \wedge f \in \mathcal{F}\}| = 1$

$WF_\tau 8$ For each point in time a group is temporally valid and has the type ALTERNATIVE or OR, it must contain at least two features:
$\forall g \in \mathcal{G} : \forall t \in \tau_\mathcal{G}(g) : \lambda_{\tau_\mathcal{G}}(g, t) \in \{alternative, or\} \implies |\omega_\tau(g, t)| \geq 2$

$WF_\tau 9$ For each point in time a group is temporally valid and has the type ALTERNATIVE or OR, it must not contain MANDATORY features:
$\forall g \in \mathcal{G} : \forall t \in \tau_\mathcal{G}(g) : \lambda_{\tau_\mathcal{G}}(g, t) \in \{alternative, or\} \implies \forall f \in \omega_\tau(g, t) : \lambda_{\tau_\mathcal{F}}(f, t) = optional$

With these well-formedness rules, we lift the rules of standard feature models to TFM. Consequently, by adhering to these rules, the state of a TFM is well-formed at each evolution step.

4.2. Evolution Paradox Classification

An evolution paradox emerges if an intermediate evolution operation is devised that violates TFM well-formedness rules in combination with already planned evolution operations for points in time after the time point at which the intermediate operation becomes effective. Consequently, the temporal order of evolution operation *planning* does not match the temporal order of evolution oper-

ation *scheduling*. For instance, in the running example of the Linux kernel (cf. Figure 4.1), the order of evolution operation planning is as follows:

t_0: Plan initial feature model.

t_1: Plan to add `AES` and `Twofish` as features of `Encfs`.

$t_{0.5}$: Replan to remove `Encfs` and add `goCrypt`.

In contrast, the temporal order of evolution operation scheduling, i.e., the points in time at which the operations should take effect, is as follows:

t_0: Plan initial feature model.

$t_{0.5}$: Replan to remove `Encfs` and add `goCrypt`.

t_1: Plan to add `AES` and `Twofish` as features of `Encfs`.

Consequently, replanning a feature-model evolution plan requires two phases: First, introducing new intermediate evolution operations (depicted by arrow ③ in Figure 4.1). Second, the intermediate operations must be incorporated as the basis for the already subsequently planned operations. However, incorporating the intermediate operations can fail as they change the basis for the subsequent operations in such a way that applying the subsequent operations leads to a structural inconsistency. For instance, in the example, the planned operation of adding `AES` and `Twofish` as children of `Encfs` can no longer be implemented as, once reaching t_1, `Encfs` will have been deleted. The reason is that the already planned operations for t_1 violate the feature-model well-formedness rules as the intermediate change at $t_{0.5}$ retroactively changed the basis the changes of t_1. Consequently, an evolution paradox has been introduced. In particular, the newly introduced group of the features `AES` and `Twofish` would not have a parent feature which violates the well-formedness rule $WF_\tau 7$.

To understand how evolution paradoxes can emerge, we analyze how an evolution operation may introduce an evolution paradox. We focus on user-level edit operations that have been identified in previous work [PDŠ12, XXJ10]. Table 4.1 lists the respective operations with respective parameters necessary for their execution and a brief description. In particular, we consider adding, removing, and moving features and groups as well as operations for changing feature and group types, and renaming features. We identified four different types of evolution paradoxes. In the following, we elaborate on each type, describe which evolution operations may lead to the respective evolution paradox, and determine which well-formedness rules are violated.

A **Non-Existent Element Edit Paradox** emerges if an evolution operation is planned modifying a feature/group (or its sub-tree), but this feature/group is deleted in an intermediate step. (violation of $WF_\tau 6$ and $WF_\tau 7$).

A **Variation Type Paradox** emerges if an intermediate edit operation results in an ALTERNATIVE or OR group that (i) contains a MANDATORY feature ($WF_\tau 8$) or (ii) has less than two child features ($WF_\tau 9$). Possible intermediate edit operations that can lead to paradoxes of this type are:

- *feature delete operations* if, in the original plan, a group is of type ALTERNATIVE or OR, and contains only the deleted and another feature at a subsequent point in time ($WF_\tau 9$).

Table 4.1.: A collection of common, basic user-level edit operations for feature models.

Operation	Description
createFeature(fid, name, gid, type)	Add a new feature to target group with a given feature id, feature name, group id, and feature type
removeFeature(fid)	Remove a feature from the feature model with a given feature id
moveFeature(fid, gid)	Move a feature to another group with a given feature id and target group id
renameFeature(fid, name)	Rename a feature with given feature id and new name
changeFeatureType(fid, type)	Change the feature variation type with a given feature id and a new type
createGroup(fid, gid, type)	Create a new group and add to a parent feature with a given parent feature id, group id, and group variation type
removeGroup(gid)	Remove a group from the feature model with a given group id
moveGroup(gid, fid)	Move a group to a new parent feature with a given group id and a new parent feature id
changeGroupType(gid, type)	Change the group variation type with a given group id and a new type

- *feature move operations* if, in the original plan, a group is of type ALTERNATIVE or OR, and contains only the moved and another feature at a subsequent point in time. As a result, only one feature remains in this group ($WF_\tau 9$). It might also happen that, in the original plan, a group type is changed to ALTERNATIVE or OR at a later point in time, and the feature that is moved in the intermediate operation into this group is MANDATORY($WF_\tau 8$).
- *feature type change operations* if the feature is contained in an AND group and in the original plan, changed to ALTERNATIVE or OR at a later point in time, while the intermediate operation changes the feature type to MANDATORY ($WF_\tau 8$).
- *group type change operations* if the intermediate operation changes the group's type to ALTERNATIVE or OR and, at a later point in time of the original plan, (i) the group's features are moved or deleted so that the group only contains one feature ($WF_\tau 8$) or (ii) a MANDATORY feature is created in or moved to that group ($WF_\tau 9$).

A **Naming Conflict Paradox** is caused by an intermediate *feature rename operation* if a feature's name is changed to a feature name that has been already introduced by subsequent operations of the original plan, e.g., by a feature creation or another feature renaming. Consequently, the names are no longer unique over the course of time ($WF_\tau 5$).

A **Transient Effect Paradox** can occur if an intermediate operation is reverted by an already existing operation scheduled for a later point in time. Thus, the effect of the intermediate operation is limited which may not be clear when devising this operation; for instance, if a feature

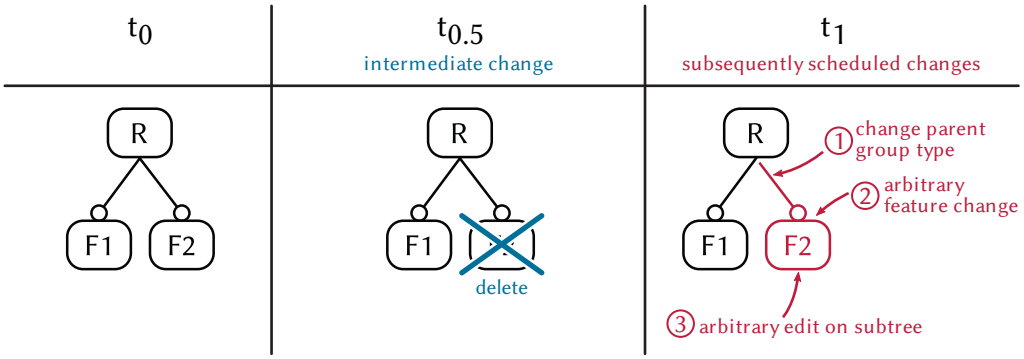


Figure 4.3.: Graphical representation under which circumstances an intermediate *delete feature* evolution operation scheduled for $t_{0.5}$ causes an evolution paradox at t_1.

is created in an intermediate operation and if the parent feature is deleted in the immediate future. These paradoxes are not violating well-formedness rules and may be introduced intentionally. However, due to the possible complexity of feature-model evolution plans, engineers should be aware of the limited effect of the intermediate operation. Relevant intermediate edit operations are:

- *feature create operations* - can be reverted by a subsequently scheduled delete operation targeting the parent group or the parent feature.
- *feature/group move operations* - the moved feature/group could become unintentionally deleted by a subsequently scheduled delete operation targeting the new parent element. Additionally, the moved feature/group could be moved again in a subsequently scheduled operation and, thus, the feature/group is only moved for a limited amount of time to the new parent structure of the intermediate operation.
- *feature/group type change operations* - a variation type change operation of an intermediate operation can become reverted by a subsequently scheduled variation type change operation on the same feature/group.

As already listed above, evolution paradoxes may be caused by various intermediate evolution operations. We identified under which circumstances an intermediate evolution operation may cause an evolution paradox. To this end, we analyze each evolution operation type defined in Table 4.1 and determinate which other subsequently scheduled evolution operations must exist in the feature-model evolution plan such that an evolution paradox arises.

Figure 4.3 visualizes in which situations an intermediate *feature delete* operation causes an evolution paradox. The arrows ①, ②, and ③ represent evolution operations scheduled for a subsequent point in time. These operations conflict with the retroactively introduced intermediate *delete feature* operation:

① A *Variation Type Paradox* occurs if a type change operation to Or or Alternative of the deleted feature's parent group has been defined previously but scheduled for a subsequent point in time and that group only contains two child features.

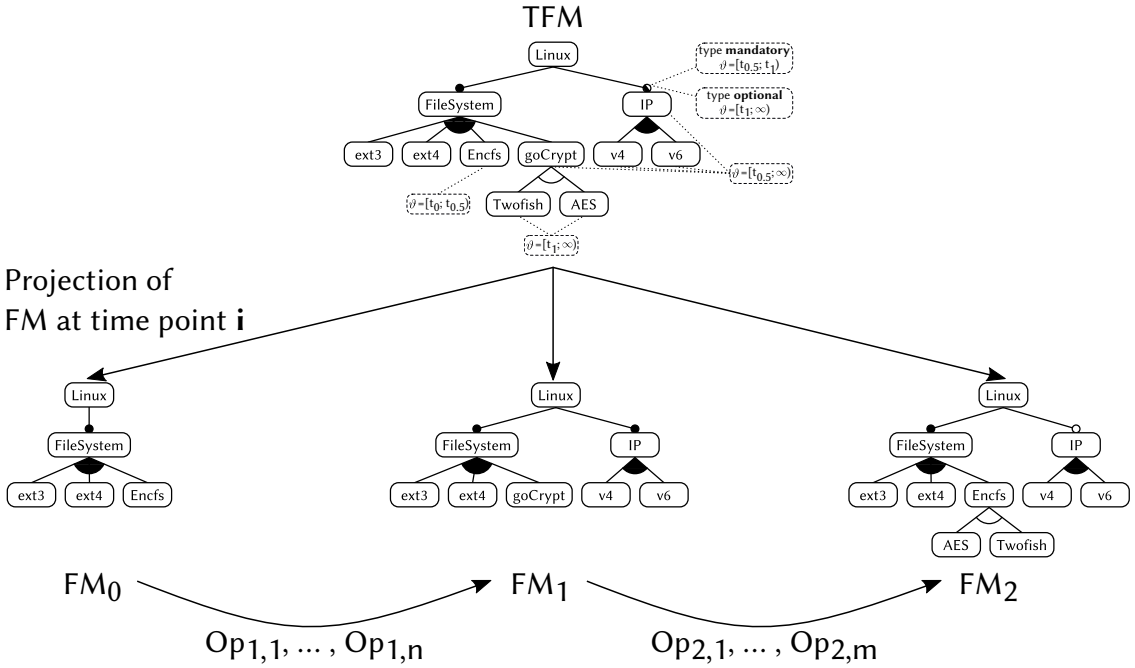


Figure 4.4.: A TFM with feature-model projections at different time points. Sequences of evolution operation lists can change transform one feature-model projection to another one.

② A *Non-Existent Element Edit Paradox* occurs if a feature change operation (e.g., change name, move, etc.) of the deleted feature has been defined previously but scheduled for a subsequent point in time.

③ A *Non-Existent Element Edit Paradox* occurs if an arbitrary edit to any element of the deleted feature's subtree has been defined previously but scheduled for a subsequent point in time.

We identified the possible situations for each evolution operation type for which an evolution paradox occurs. The respective graphical representations and descriptions can be found in Appendix B. We use this information to derive preconditions that must hold such that an evolution operation does *not* introduce an evolution paradox.

4.3. Ensuring Consistent Feature-Model Evolution Plans

Figure 4.1 highlights the challenges of consistent feature-model evolution planning and replanning. An evolution plan in a TFM is free from evolution paradoxes if each evolution operation that modifies the TFM does not violate the TFM well-formedness rules. However, verifying all well-formedness rules for each point in time of a TFM is inefficient. Thus, we reason about the impact of evolution operations on a feature model and analyze whether they would cause an evolution paradox. To this end, we use Structural Operational Semantics (SOS) and model all evolution operations as SOS rules on simple feature models. This is semantically equivalent to TFMs but enables us to define reason on evolution operation level instead of TFM state level. Figure 4.4 shows the relation between a TFM, its projected feature models for certain time points, and evolution operations. From a TFM, feature models (FM_i) at a point in time i can be projected. A list of evolution operations ($Op_{i+1,1}, ..., Op_{i+1,n}$) may transform a feature-model projection from one time

point (FM_i) to a feature-model projection of the subsequent evolution step (FM_{i+1}). Thus, a feature model with a sequence of feature-model evolution operation lists is semantically equivalent to a TFM. In particular, a TFM expresses multiple evolution *states* of a feature model and evolution operations define how to transform one feature-model evolution state to another. As a basis for the SOS rules, we first formalize feature-model evolution plans.

4.3.1. Formalizing Feature-Model Evolution Plans

For simplicity of notation, we formalize feature-model evolution plans in terms of modifications (i.e., evolution operations) to a simple feature model. A feature-model evolution plan consists of an ordered list of evolution operations on a feature model that are scheduled for different points in time. In Definition 4.2, we provide a formalization of a feature-model evolution plan [HNS+20]. Such a plan consists of a feature model FM and an ordered list of planning sections *Sections*. An FM is defined by its root feature with the ID *RootFeatureID* and a feature table FT that is used to capture the hierarchy, relations, and properties of features and groups. Each planning section of the list of *Sections* contains an ordered list of evolution operations and a point in time for which the evolution operations should be applied. We use a natural numbers model of time, i.e., each concrete time point of the planning sections is mapped to a $t_i \in \mathbb{N}$ whereas the order of these numbers preserves the order of original time points.

Definition 4.2: Feature Model Evolution Plan

A feature-model evolution plan $Plan = (FM, Sections)$ is defined as

(i) an initial feature model $FM = (RootFeatureID, FT)$ with *RootFeatureID* being the ID of the root feature and FT is a feature table representing the tree hierarchy,

(ii) feature table FT entries are structured as $[FeatureID \mapsto (Name, ParentFeatureID, \overline{Groups}, FType)]$ with *FeatureID* a feature ID, *Name* the name of that feature, *ParentFeatureID* the ID of the parent feature, $\overline{Groups}$ a set of child groups, *FType* feature variation type,

(iii) a group is defined as a tuple $g = (GroupID, GType, \overline{Features})$, with the group ID *GroupID*, the group variation type *GType*, and $\overline{Features}$ is a set of child feature IDs, and

(iv) an ordered list $Sections = \{section_1, ..., section_n\}$ of *planning sections* that each contain all evolution operations for the same time point $t_i \in \mathbb{N}$, with $section_i = (t_i, Ops_i)$, where $Ops_i = (Op_{i,1}, \ldots, Op_{i,n})$ is an ordered list of n evolution operations (operations defined as in Table 4.1).

A consistent feature-model evolution plan does not contain two planning sections for the same point in time. Additional evolution operations can be associated with arbitrary existing or new planning sections. This enables replanning of feature-model evolution by retrospectively integrating evolution operations as intermediate planning steps.

4.3.2. Paradox-Free Execution Semantics for Feature-Model Evolution Plans

To ensure that no evolution paradoxes are introduced into a TFM, we model the effect of evolution operations in terms of execution semantics on the basis of the formalized feature-model

evolution plans [HNS+20]. This execution semantics prevents the introduction of evolution paradoxes before they violate TFM consistency. To this end, we use rewriting logic to guarantee that the execution of each evolution operation of a feature-model evolution plan does not violate any TFM well-formedness rules (cf. Section 4.1).

We refer to the planning section that contains a currently processed evolution operation as the *active planning section*.

Definition 4.3: Active Planning Section

We define an active planning section *ActiveSection* for a feature-model evolution plan $Plan = (FM, Sections), FM = (RootFeatureID, FT), Sections = \{section_i = (t_i, Ops_i) | i = 1, \ldots, n\}$ as

$ActiveSection = (Op_{t_i,1}, \ldots Op_{t_i,n}), t_i = t_{cur}$ with t_{cur} being the current time. At most one planning section within an feature-model evolution plan can be active at a given time $t_{cur} \in \mathbb{N}$.

With the definition of an active planning section, we define a planning state of a feature-model evolution plan by the current time, an active planning section, and an ordered list of remaining planning sections.

Definition 4.4: Planning State

A planning state *PlanningState* of a feature-model evolution plan $Plan = (FM, Sections)$ with an active planning section *ActiveSection* at the current time $t_{cur} \in \mathbb{N}$ is defined as

$PlanningState = (FM, t_{cur}, ActiveSection, RemainingSections)$, where *RemainingSections* is as an ordered list of all remaining planning sections scheduled for later time points than t_{cur} in *Sections*.

Structural Operational Semantics for Evolution Operations

We specify the behavior of the considered feature-model evolution operations (cf. Table 4.1) using SOS. To illustrate the SOS rules, we use the create feature operation. Appendix B contains the entire set of rules. Each SOS rule describes pre-conditions (above the line) and a transition from one state to another (below the line). The transition can only be executed if all pre-conditions are satisfied.

Semantics Rule 4.1 defines the semantics of a feature create operation of a feature with the ID *FeatureID*, the name *Name*, and the type *FType* that should be added as child feature to a group with ID *ParentGroupID* that has a parent feature with ID *ParentFeatureID*. We define the following auxiliary notions that enable us to formalize the preconditions:

- $isUniqueName(Name, FT)$ checks whether the name *Name* is unique in the feature table *FT*
- $addFeatureToGroup(FT, ParentGroupID, FeatureID)$ returns a modified feature table FT' in which the feature ID *FeatureID* is added to the set of child feature IDs $\overline{Features}$ of a group $g = (ParentGroupID, GType, \overline{Features})$.
- $isValidType(FT, FeatureID)$ checks whether the type of the feature with ID *FeatureID* is valid, i.e., for a Mandatory feature, it is only valid if the feature is in an And group or if it is the root feature.

The feature create operation operation can only be executed if and only if the pre-conditions above the line hold, i.e.:

- the given *FeatureID* is unique,
- no other feature is assigned the name *Name* ($WF_\tau 3$),
- the parent group with ID *ParentGroupID* exists ($WF_\tau 6$),
- the given variation type *FType* of the new feature is valid within the resulting FM ($WF_\tau 1$, $WF_\tau 8$ and $WF_\tau 9$)

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature is added as new child feature of the specified group and a new feature table entry $[FeatureID \mapsto (Name, ParentFeatureID, \emptyset, FType)]$ is added to FT with an empty set for the child group IDs.

Semantics Rule 4.1 (Feature Create Operation)

$$\frac{\begin{array}{c} FT(FeatureID) = \bot \\ isUniqueName(Name, FT) \\ isValidType(FT'', FeatureID) \\ FT' = addFeatureToGroup(FT, ParentGroupID, FeatureID) \\ FT'' = FT' + [FeatureID \mapsto (Name, ParentFeatureID, \emptyset, FType)] \end{array}}{\begin{array}{c} FM(RootFeatureID, FT) \\ \textbf{\textit{createFeature}}(FeatureID, Name, ParentGroupID, FType) \\ \Rightarrow \\ FM(RootFeatureID, FT'') \end{array}}$$

Structural Operational Semantics for Feature-Model Evolution Plans

Based on the semantics of feature-model evolution operations defined in SOS, we define a paradox-free execution semantics for feature-model evolution plans. To this end, we define three rules that are applied depending on the planning state of the evolution plan.

Semantics Rule 4.2.1 is applied if the planning state's active planning section still contains feature-model evolution operations. In the precondition, it is checked whether the next operation to be executed ($Op_{cur,1}$) does not introduce evolution paradoxes using the semantics rule of that particular operation. If this is the case, the operation is removed from the active planning section (below the line) and the operation is executed resulting in the modified feature table FT'. If the pre-conditions are not met, an evolution paradox would be introduced and the execution aborts. We denote an empty list of remaining evolution operations within the active planning section as ϵ.

Semantics Rule 4.2.1 (Executing Feature-Model Evolution Operations)

$$\frac{\begin{array}{c} FM(RootFeatureID, FT)\ Op_{cur,1} \\ \Rightarrow \\ FM(RootFeatureID, FT') \end{array}}{\begin{array}{c} (t_{cur}, FM(RootFeatureID, FT)\ Op_{cur,1}, \ldots, Op_{cur,m_{cur}}, \overline{RemainingSections}) \\ \Rightarrow \\ (t_{cur}, FM(RootFeatureID, FT')\ Op_{cur,2}, \ldots, Op_{cur,m_{cur}}, \overline{RemainingSections}) \end{array}}$$

Semantics Rule 4.2.2 advances the current time t_{cur} to t_{next} if no evolution operation is left in the active planning section. The subsequent planning section is then set as the active planning section.

Semantics Rule 4.2.2 (Advancing Time)

$$\frac{t_{cur} < t_{next}}{(t_{cur}, FM(\mathit{RootFeatureID}, FT)\ \epsilon, (t_{next}, \overline{Ops}); \overline{\mathit{MoreSections}})} \Rightarrow (t_{next}, FM(\mathit{RootFeatureID}, FT)\ \overline{Ops}, \overline{\mathit{MoreSections}})$$

Semantics Rule 4.2.3 defines the successful execution of the entire feature-model evolution plan if all evolution operations of all planning sections were successfully applied. We denote an empty list of remaining planning sections within a planning state with ρ.

Semantics Rule 4.2.3 (Fully Processed Plan)

$$\frac{}{(t, FM(\mathit{RootFeatureID}, FT)\ \epsilon, \rho)}$$

In Theorem 4.1, we formulate that the defined execution semantics ensures structural consistency of a feature-model evolution plan and, thus, guarantees that the resulting TFM is free from evolution paradoxes.

Theorem 4.1: Structural Consistency of a Feature-Model Evolution Plan

Let $\mathit{Plan} = (FM, \mathit{Sections})$ be a feature-model evolution plan with an initial feature model $FM = (\mathit{RootFeatureID}, FT)$ and a non-empty list of planning sections $\mathit{Sections} = (t_1, \overline{Ops_1}); (t_2, \overline{Ops_2}); ...; (t_n, \overline{Ops_n})$. Then, Plan is structurally consistent if and only if the execution of all planning sections on FM according to our above defined semantics terminates:

$$(t_0,\ FM(\mathit{RootFeatureID}, FT)\ \epsilon,\ \mathit{Plan}) \overset{*}{\Rightarrow} (t_n,\ FM(\mathit{RootFeatureID}, FT')\ \epsilon,\ \rho)$$

The idea to prove Theorem 4.1 is to use structural induction over the previously defined semantics rules. If a semantics rule of an evolution operation can be applied, i.e., its preconditions are met, no paradox is introduced by definition. The remaining rules remove the applied operations of planning sections and advance time and, thus, removes empty planning sections until no planning section remains. However, if a paradox would be introduced by an evolution operation, the respective rule's preconditions are not fulfilled and, thus the rule cannot be applied and the active planning section cannot advance.

In summary, we are now able to define feature-model evolution plans in terms of an initial feature model with an ordered set of planned evolution operations. As Figure 4.4 illustrates is a feature-model evolution plan equivalent to a TFM. To guarantee that replanning of such evolution does not introduce evolution paradoxes, we defined the execution semantics on basis of evolution operations. As a consequence, when modifying a TFM, all planned changes are captured in terms of evolution operations and analyzed using the above-mentioned method. If no of the modifying operation introduces an evolution paradox, the evolution operations are used to modify the TFM accordingly.

4.4. Evaluation

In this chapter, we address challenges for planning and replanning feature-model evolution by providing a method to ensure evolution paradox free replanning. We seek to answer the first part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention** with that contribution and evaluate our method by three means: first, we show feasibility by providing an implementation that is integrated in the tool suite DarwinSPL; second, we inspect scalability of our method; finally, we empirically analyze whether both researchers from academia and practitioners from the industry acknowledge the need for methods to detect evolution paradoxes and whether our method seems suitable.

4.4.1. Implementation

We implement the method to guarantee paradox-free feature-model evolution planning of TFMs within the tool suite DarwinSPL. The semantic rules are implemented using the rewriting logic system Maude [CDE+07]. The way we defined our semantic rules requires knowledge about the performed evolution operations. However, DarwinSPL only saves and maintains TFMs and, thus, does not save evolution operations. To make DarwinSPL compatible with our Maude implementation, we need to be able to capture and execute evolution operations in DarwinSPL, and to formalize a TFM and its evolution operations. In the following, we elaborate on the implementation of our method. First, we describe how we capture and execute evolution operations. Second, we explain the structure of our Maude implementation, and how we connect DarwinSPL with Maude.

Capturing and Executing Evolution Operations As DarwinSPL only creates and maintains TFMs, it does not save the performed evolution operations. However, in our execution semantics, we only detect evolution paradoxes by checking formalized evolution operations. Consequently, we extend DarwinSPL to also save the performed evolution operations. In particular, we create a metamodel to capture the different evolution operations. Figure 4.5 shows an excerpt of this metamodel. An `OperationModel` holds a set of `EvolutionOperations`. Each `EvolutionOperation` has an `operationDate` which corresponds to the point in time for which the operation is scheduled and a `commandStackIndex` which defines the order of operation definition (cf. Section 4.2, distinction between temporal order of evolution operation *devision* and *scheduling*).

We distinguish between evolution operations that affect groups, i.e., `GroupOperations`, and operations that affect features, i.e., `FeatureOperations`. Both reference the affected group or the affected feature, respectively. The concrete operations inherit from `GroupOperation` or `FeatureOperation`. For brevity, Figure 4.5 only shows an excerpt of the operations. We modeled each operation defined in Table 4.1.

As some operations have similar effects, we modeled them as compound operations with shared basic operations that are not performed by users. For instance, a `FeatureDelete` operation removes the feature from its current group and sets the temporal validity of the feature to end at the operation's date. A `FeatureMove` operation removes the feature as well from its current group and adds it to another. The `FeatureDelete` and `FeatureMove` operations share to remove the considered feature from its current group. Thus, we modeled a reusable `FeatureDetach` operation that performs this task and added this operation as sub-operation to the `FeatureDelete` and the `FeatureMove` operation.

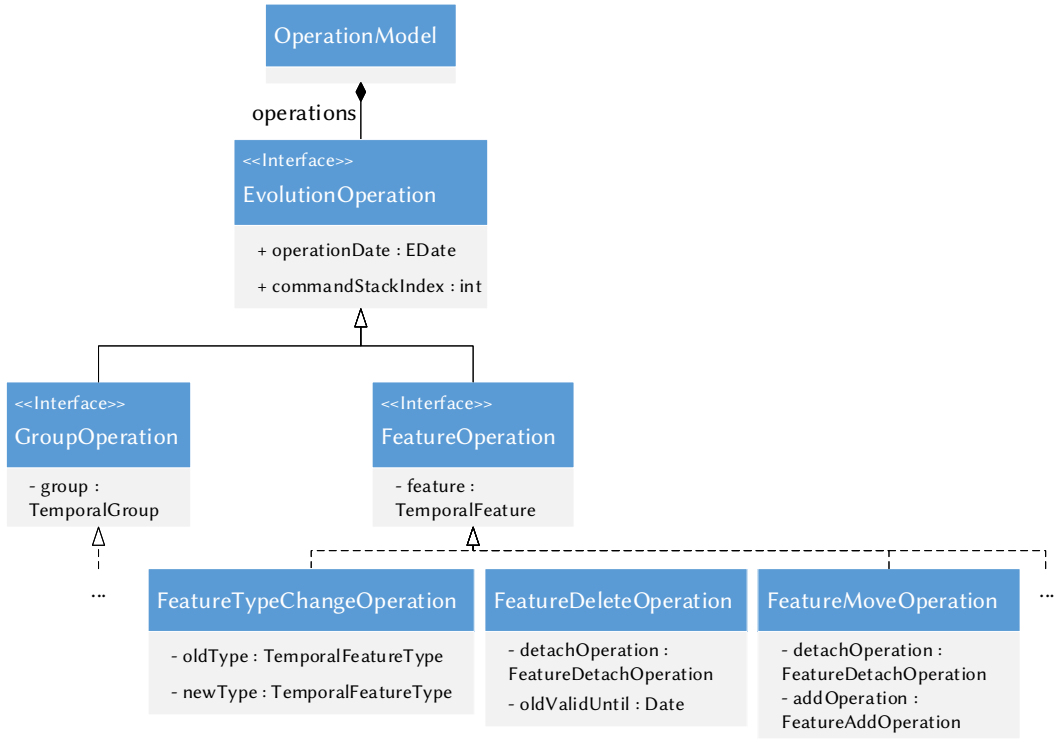


Figure 4.5.: Excerpt of metamodel for feature-model evolution operations.

Some operations need to reference certain TFM elements. For instance, the `FeatureTypeChangeOperation` has two references to `TemporalFeatureType`. One reference for the new type and one for the old one. For paradox detection, we do not need the information on the old type but we need it to provide an *undo* functionality in the editor.

For the sake of reusability and modularity, we separated the execution of the evolution operations from the editor and followed the Model-View-Controller design pattern [Gam95]. Consequently, multiple editors (*views*) exist that create operations and pass them to an `OperationInterpreter` (*controller*). The operation interpreter is responsible for actually modifying a TFM (*model*) based on the provided evolution operations and storing the operations in an `OperationModel`. Before modifying a TFM, the `OperationInterpreter` provides the possibility for external analysis methods to analyze the new operations. To this end, we implement an extension mechanism that enables to register as evolution operation analyses extension. Thus, before executing an operation, a registered extension can analyze the TFM and the new operation, and can stop the operation execution.

Detecting Evolution Paradoxes To guarantee that a feature-model evolution plan is paradox-free, we implemented the semantics rules using the rewriting logic system Maude [CDE+07]. To check whether an intermediate operation in a TFM performed in DarwinSPL leads to a paradox, we need to translate the TFM with all already performed evolution operations as Maude as input. In DarwinSPL, we devise an `EvolutionChecker` component that is responsible for translating all necessary information as input for Maude and for translating the answer of Maude back to an understandable form for end users. The `EvolutionChecker` registers itself as evolution operation

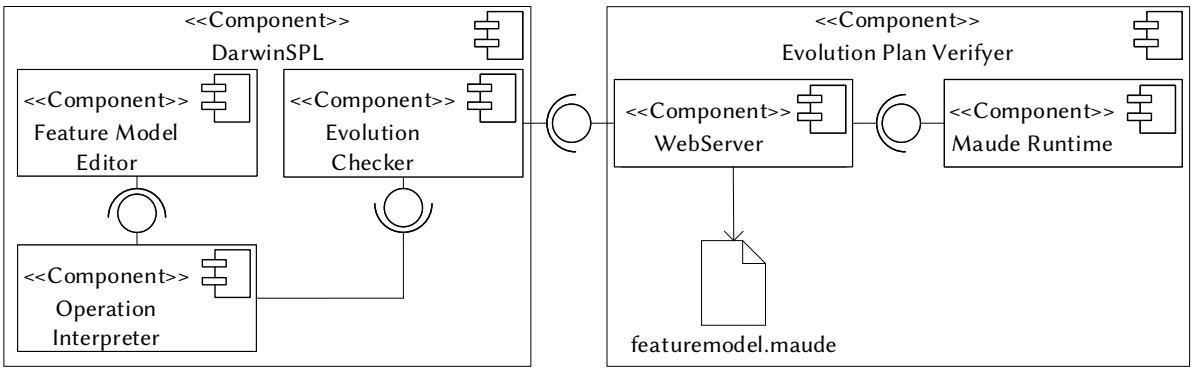


Figure 4.6.: Components involved in detecting evolution paradoxes for DarwinSPL.

analyses extension at the `OperationInterpreter`. Thus, it is called if a new operation is to be executed and can trigger the evolution paradox detection.

Figure 4.6 shows the components that are involved in modeling TFMs and detecting evolution paradoxes. Detecting evolution paradoxes for large feature models can be computationally very expensive. Thus, we define the structure of our components in such a way that components for modeling a TFM are separated from components for detecting evolution paradoxes. In particular, we devise a RESTful web interface that enables to outsource the paradox detection to another system, e.g., to a powerful mainframe. To this end, we devise a `EvolutionPlanVerifyer` component that contains a `WebServer` component providing the web interface. The `WebServer` starts a new Maude instance, and forwards the formalized TFM and evolution operations. In the Maude module `featuremodel.maude`, we implement the formalization of the feature models and the execution semantics which are used to detect evolution paradoxes. Thus, the `EvolutionPlanVerifyer` does not have to run on the same system of an end-user that is responsible for modeling a TFM. Another benefit is that this structure enables us to deploy the `EvolutionPlanVerifyer` on non-Unix systems despite Maude only being available for Unix systems. To this end, the `EvolutionPlanVerifyer` can be deployed in a virtual machine or container (e.g., Docker).

The goal of our implementation is to inform users of evolution paradoxes that a specific editor operation causes. Figure 4.7 show the workflow of the previously introduced components. First, the editor creates a new operation object and passes it to the `OperationInterpreter` which saves it in an operation model. Then, the `EvolutionChecker` is instructed to check for paradoxes. It collects and sorts the operations of the operation model based on their `operationDate`, i.e., the date for which the operations are scheduled. Subsequently, it translates the TFM and all operations as formalized input for Maude and sends it to the `WebServer`. Thus, the entire feature-model evolution plan as well as the newly devised evolution operation is encoded as one formula for Maude. For brevity, we omit the `WebServer` from Figure 4.7. Using the execution semantics we specified, Maude searches for evolution paradoxes and returns the result via the `WebServer` to the `EvolutionChecker`.

If an evolution paradox is detected, the `EvolutionChecker` shows an error dialog via the editor which states that the triggered evolution operation would lead to an evolution paradox. For instance, in our running example (cf. Figure 4.1), the feature `Encfs` is deleted in an intermediate evolution step resulting in an evolution paradox in a subsequent evolution step. Figure 4.8 shows

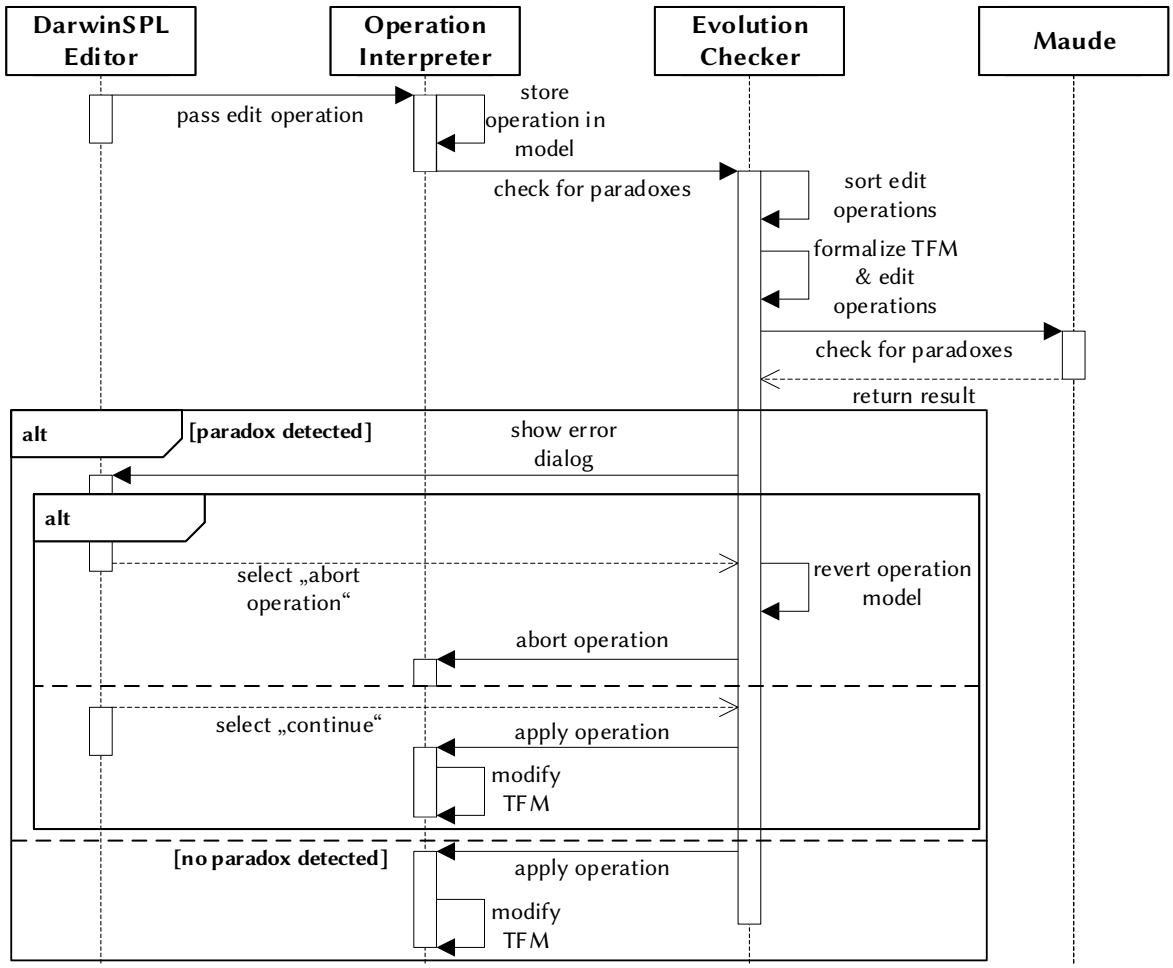


Figure 4.7.: Workflow of the DarwinSPL components to check for evolution paradoxes after performing an editor operation.

the error dialog if a user tries to execute this intermediate operation in DarwinSPL. It shows that a `addGroup` operation should be executed (at the bottom of the error message) but the parent feature to which it should be added, i.e., `Encfs`, does not exist at the analyzed time point. Users may decide whether to abort the operation execution or whether it should be applied despite it will cause a paradox, e.g., to fix it manually. If no evolution paradox is detected or if the user selects to execute an operation that leads to a paradox, the `OperationInterpreter` modifies the TFM based on the devised operation. Thus, with DarwinSPL and the semantics rules encoded in `Maude`, we are able to plan and replan feature model evolution without accidentally introducing evolution paradoxes. Note that we currently, detect only the first evolution paradox that exists in the feature-model evolution plan and Maude stops applying the execution semantics of evolution operations planned for subsequent time points. Thus, a sensible future extension would be to define execution semantics that proceeds to apply operations and, consequently, finds more potential evolution paradoxes that exist in the feature-model evolution plan.

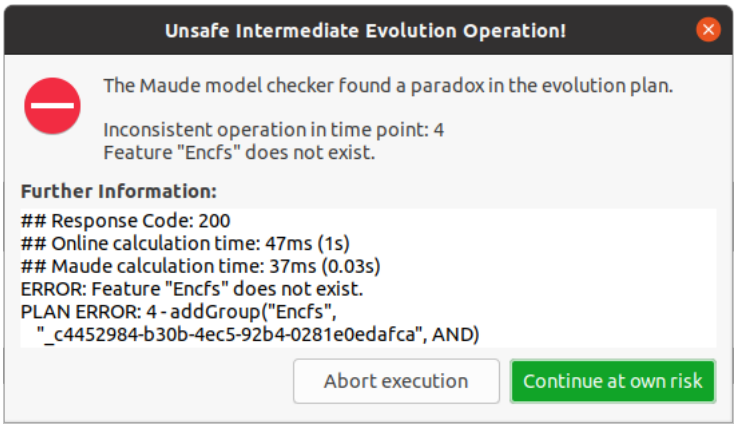


Figure 4.8.: Screenshot of the error message of DarwinSPL stating that the intermediate operation in the Linux kernel feature model of deleting `Encfs` at $t_{0.5}$ would lead to a paradox.

4.4.2. Scalability Evaluation

The answer to **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention** depends on the practicability and, thus, on its scalability. When replanning the evolution of a feature model, typically multiple intermediate operations are devised. Additionally, multiple replanning steps may be introduced. However, it is time-intensive and, thus, expensive if evolution paradoxes are not directly detected after devising an intermediate evolution operation. For instance, multiple intermediate operations may lead to several evolution paradoxes and also may interact. Consequently, it may happen that multiple (intermediate) evolution operations must be reverted. To provide a remedy, immediate detection of evolution paradoxes after devising an intermediate operation is necessary. Consequently, it is crucial that our method scales even to large feature models that are used in the industry. We pose an additional research question that addresses this scalability and contributes to answering **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**:

RQ2.1 Does the prevention of evolution paradox introduction scale in terms of performance?

To provide an answer to **RQ2.1** we evaluate the scalability of our method by analyzing its performance based on multiple existing feature-model evolution scenarios.

Setup We use a set of existing feature-model evolution scenarios as the basis of our evaluation. In particular, we use four small and medium-sized feature models and their well-documented evolution history [NLS18]. In particular, the evolution scenarios base on the feature models of the following product lines: *Mine Pump* [KMS+83], *Wiper* [HRR+11], *Vending Machine* [Cla10], and *Body Comfort System* [LLL+13]. Additionally, we use a real-world large-scale feature model and its evolution of a financial company that is accessible in the FeatureIDE online repository[1]. We modeled each feature-model evolution scenario as one TFM using DarwinSPL. Table 4.2 shows the number of features, groups, planning sections (i.e., evolution steps), and evolution operations for each evolution scenario.

[1] `https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples/FeatureModels/FinancialServices01`

Table 4.2.: Properties of the feature model evolution scenarios used for the evaluation.

	Overall Features	Overall Groups	Planning Sections	Edit Operations
Mine Pump	9	3	3	36
Wiper	14	5	4	60
Vending Machine	19	7	6	107
Body Comfort System	48	16	5	207
FinancialServices01	1,083	259	10	3,785

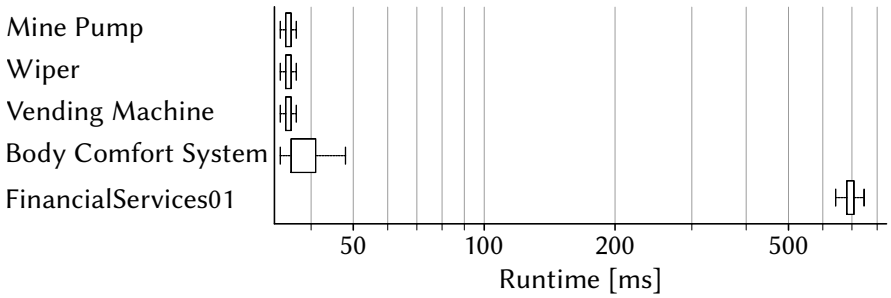


Figure 4.9.: Runtimes of our execution semantics after inserting one evolution operation in an intermediate step (100,000 repetitions, outliers omitted).

As the original data sets do not contain intermediate evolution steps and, thus, no paradoxes, we simulate this by generating random evolution operations as intermediate steps for each scenario. In particular, we randomly select an existing evolution step (except for the last one), and randomly generate and apply a new evolution operation (cf. Table 4.1). After inserting an evolution operation, we analyze the entire feature-model evolution using our execution semantics and measure runtime. We repeat this procedure 100,000 times for each evolution scenario whereas we always use the original unmodified feature-model evolution plan as the basis.

Results Figure 4.9 shows the runtimes for each added intermediate operation grouped by the respective scenario. We omit outliers for better readability. Table 4.3 summarizes the relevant information in terms of average and maximum runtime. For the small and medium-sized scenarios, we are able to analyze the entire feature-model evolution plan in **less than 300 ms** and on the **average in less than 40 ms**. For our large-scale industry scenario (i.e., *FinancialServices01*), each analysis was performed in **less than 3 seconds** and on the **average in less than 700 ms**.

We further investigated whether specific evolution operations require more runtime to analyze than others. Figure 4.10 shows an overview of the average runtime for each evolution operation type that has been applied to the *FinancialServices01* scenario. The maximum difference between the operations that are the fastest to be analyzed (i.e., *Change Feature Type*) and the operations that take the most time to be analyzed (i.e., *Delete Group*) is **63.9 ms** and, thus, **less than 10%**. Consequently, our method is stable against the type of operation that is applied.

In summary, we deem the runtimes as adequate for practical application and, thus, can answer **RQ2.1** positively. When defining a new feature model, evolution plans potentially change frequently and, thus, many intermediate operations are defined. However, for such small fea-

Table 4.3.: Average and maximal runtimes of our execution semantics after inserting one evolution operation in an intermediate step (100,000 repetitions) and percentage of operations that lead to an evolution paradox.

	Average Runtime	Maximum Runtime	Paradoxical Operations
Mine Pump	36 ms	249 ms	23.49 %
Wiper	36 ms	254 ms	36.75 %
Vending Machine	36 ms	247 ms	31.01 %
Body Comfort System	38 ms	255 ms	24.60 %
FinancialServices01	692 ms	2,764 ms	36.17 %

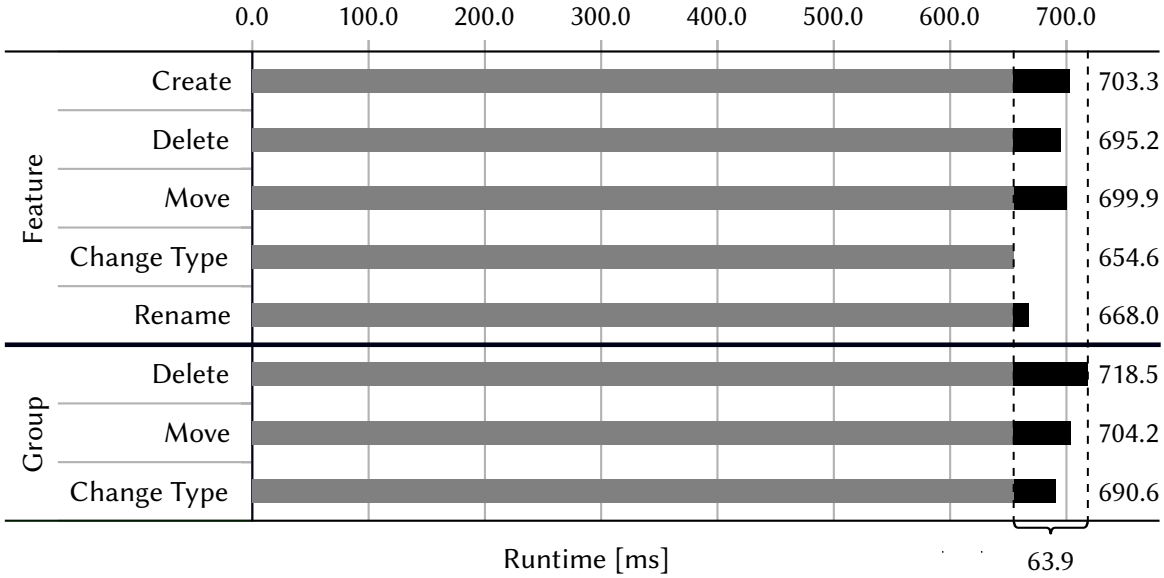


Figure 4.10.: Average runtime of our execution semantics for the *FinancialServices01* scenario, grouped by the evolution operation type that was inserted as intermediate planning step.

ture models, our method analyzes each operation in less than 300 ms and, thus, does not impede the usage of feature-modeling tools. For large feature models, our analyses partially require a few seconds. However, large feature models are typically more stable and are only changed a few times in a week (cf. Section 4.4.3). Regarding the significant size of such scenarios, we expect high acceptance of this amount of runtime.

Table 4.3 also lists the percentage of generated intermediate operations that cause an evolution paradox if applied. The results show that **23%-37%** of the intermediate operations lead to an evolution paradox. The fact that about 1/3 of the potential operations would lead to structural inconsistencies that might be noticed only at a later point in time shows the urgency of an automated evolution paradox detection mechanism.

Threats to Validity

Our scalability evaluation is subject to threats to validity. The external validity is threatened as we only use one real-world feature model evolution scenario. The other scenarios partially base on real-

world feature models, but the evolution was artificially constructed. The *FinancialServices01* dataset is one of the largest real-world feature models that are freely accessible and which covers the most evolution steps. However, data on real-world feature-model evolution is nearly non-existent. Thus, our measurements base on one of the most representative data set that is accessible. Additionally, the evolution of the other scenario has not been performed by us but in another study in which the goal was to define realistic evolution scenarios for existing feature models.

The external validity is additionally threatened as we randomly generate evolution operations in randomly selected intermediate steps. Consequently, it might happen that uncommon operations, such as deleting a key feature, might be executed and the intermediate steps might be introduced at very uncommon points in time, such as for a very old version with a lot of planned evolution steps afterwards. Even if we include non-representative operations, we assume that we also cover typical intermediate operations as we use heavy repetition of the random generation (100,000 repetitions). The previously mentioned assumption and the fact that we have very homogeneous results indicate that the results are representative. More importantly, very uncommon operations, such as deleting a key feature, might also lead to more disruptive changes that require more computation time. In summary, we argue that real-world intermediate operations are within the distribution of runtimes which does not change our interpretation of the results.

The internal validity might be affected as we reconstructed the existing evolution scenarios in DarwinSPL. For all scenarios except for *FinancialServices01*, we manually defined the evolution scenarios as described in the original publication [NLS18]. Afterwards, we verified correct modeling by comparing the final TFM with the original feature model versions. For *FinancialServices01*, we relied on the automatic import functionality for multiple feature model versions of DarwinSPL. This has two reasons. First, it is not feasible to remodel such a large model in a sensible amount of time. Second, no documentation on the evolution exists and, thus, we would have to guess the actual evolution operations which would be similar to the automatic import of DarwinSPL. As a result, DarwinSPL reproduces the evolution operations that lead to the least number of necessary operations to capture the different versions. However, this might lead to falsely detected operations and uneven operation distributions. For instance, DarwinSPL detected 9,639 feature type change operations but no group move operations during the import. As we do not have access to the original operations, we do not know whether this matches the performed operations or whether it differs. Even if we falsely imported operations, the randomly generated intermediate operations only lead to evolution paradoxes if the structure of a future feature-model version would be inconsistent. Thus, it does not matter whether we detected the correct operations as long as they result in the same structure. We verified that the import mechanism correctly imported the versions be comparing the original versions with the state of the imported TFM at the respective time points.

4.4.3. Empirical Assessment

To estimate the usability of our method, we evaluate whether researchers from academia and practitioners from industry acknowledge the problem of evolution paradoxes and whether they assess our solution as suitable. In particular, we conducted semi-structured interviews with three different industry partners and sent an anonymous online survey to experts from academia in the field of SPL research. To answer **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**, we need to understand the current state-of-practice, which problems arise in practice

regarding inconsistencies, and whether our method would help to provide a remedy. We pose four additional sub-research questions that we answer in this section to answer **RQ2**:

RQ2.2 What is the current state-of-practice in SPL evolution planning?

RQ2.3 How suitable are feature-model evolution plans to plan SPL evolution?

RQ2.4 Do evolution paradoxes in feature-model evolution plans occur, and are they problematic?

RQ2.5 How valuable is an automated mechanism to detect evolution paradoxes?

In the following, we describe the details of our interviews and online survey, and, subsequently, we elaborate on the results by answering the previously stated research questions.

Procedure In this paragraph, we describe our procedure to capture data for our empirical evaluation. The first part describes the semi-structured interviews with our industry partners and the second part describes the online survey we devised for experts from academia.

Semi-Structured Interviews with Industry Partners We conducted semi-structured interviews with three industry partners from different business domains and company sizes. They all share the usage of feature models to capture the variability of their product lines. The first industry partner is a small *web-application company* that develops their applications mainly using feature-oriented development. Recently, they also deal with document generation based on product lines. Thus, their feature model consists of several text paragraph features that can be used to compose an entire document, mainly legal texts. The second industry partner is a *financial company* with over 6,000 employees. A small team models properties of credits using a feature model, e.g., credit periods or interest rates. The third company is an *automotive company* that manufactures cars and has hundreds to thousands of developers contributing to multiple SPLs. Most interestingly, this company does not use feature models for active development, but a custom solution. However, the department responsible for software variability recently started to employ feature models to perform several analyses.

We devised a guide that we used to structure the interviews to retrieve comparable results. The entire guide can be found in Section C.1. The interview guide is structured into two parts. The first part addresses the general usage of feature models as well as the planning of SPLs using feature models. In the second part, we pose questions regarding evolution paradoxes and our method to prevent their introduction. We conducted all interviews via an online video conferencing platform and documented the discussions in writing.

Online Survey with Experts from Academia To incorporate the state-of-art in research, we asked researchers in the domain of SPL engineering and feature-oriented development to participate in an anonymous online survey. Section C.2 shows the questionnaire that we used in our survey. Similar to the interviews, the questionnaire is structured in two sections: one addressing general feature-model planning and one addressing evolution paradoxes. In total, the online survey consists of six questions and additionally two open text fields for comments and feedback (one after each section). We sent the survey to 27 researchers from academia who deal with SPLs and feature modeling in their research. Additionally, we asked them to send this survey to other researchers who might be interested in it. In total, 19 researchers participated.

Results In the following, we describe the results of the interviews and the online survey. We structure this evaluation by successively answering **RQ2.2–RQ2.5**. For each research question, we discuss the results from the interviews and the survey separately, and, subsequently, give a summary.

RQ2.2 *What is the current state-of-practice in SPL evolution planning?*

Semi-Structured Interviews with Industry Partners To answer this question, we asked our interview partners in which way they are personally involved in planning product-line evolution, how this is done in their company, and which factors drive the evolution. In the *automotive company*, SPL evolution is planned based on specification sheets and project management systems that both are not specific to product lines or feature models. The planning in this company is performed on management level, and the resulting specification sheets are distributed to the different development departments that implement single features. Typically, the planning is done once on management level when the newly added features for a new car are determined, and if the implemented features are integrated, the variability is captured in a specific spreadsheet. In the *web application company*, evolution is performed on short notice and, typically, long-term plans do not exist. The evolution is planned from sprint to sprint using an agile development method. In the *financial company*, feature models have been introduced recently before the interview. Nonetheless, they already plan the evolution using multiple versions of a feature model. In particular, a "target" feature model is devised which contains features that should be integrated at a certain point in time. Active development is then performed using the "current" feature model. Typically, the "current" model is changed every two weeks. As a result, the goal is to successively change the "current" feature model to match the "target" feature model while still allowing to perform additional operations. This procedure matches our method in which we devise evolution operations for a future point in time and allowing to perform intermediate operations. However, the consistency maintenance between the "current" and the "target" feature model is manual in the process of the financial company.

Online Survey with Experts from Academia We deliberately did not pose a question in the online survey regarding **RQ2.2** as our goal is to assess the current state-of-practice in industry. Consequently, we cannot assume to retrieve representative data from the questionnaire regarding this research question. However, one of the participants used an open comment field to provide an assessment of that topic. The participant argues that employees who are responsible for SPL planning are typically dealing with requirement sheets and project management systems, but not with feature models or other technical SPL artifacts.

The results from the interviews show that the planning methods strongly depend on the size of the company and of the number of involved developers and, thus, there is not one simple answer to **RQ2.2**. For large companies, hierarchical planning is typically used in which the implementing developers might not even know about the product line itself, whereas for medium-sized or small development teams, the developers are directly involved in the planning. Additionally, the level of integration of feature modeling plays a pivotal role on the degree of formalism of the used planning method. Thus, if feature models are already important artifacts in the development

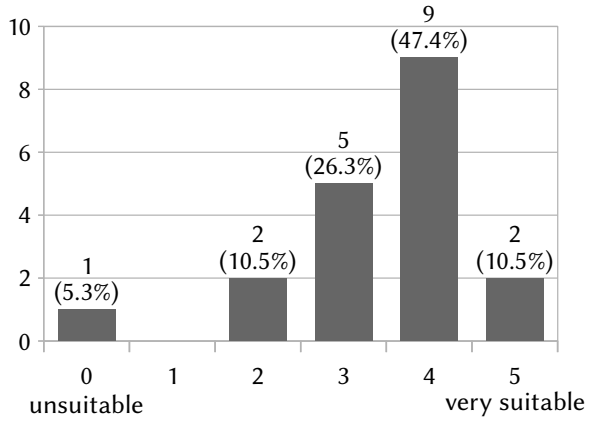


Figure 4.11.: Estimated suitability of feature-model evolution plans for SPL planning in **small** teams from the online survey with experts from academia.

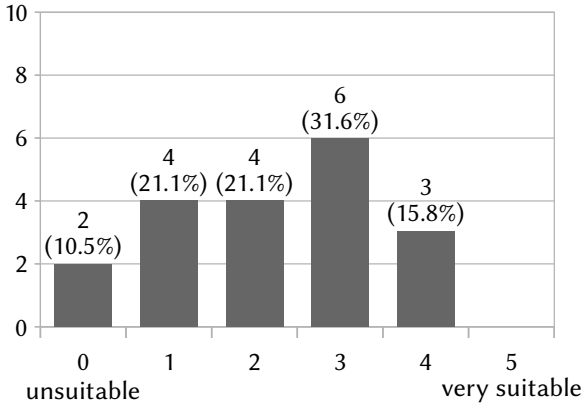


Figure 4.12.: Estimated suitability of feature-model evolution plans for SPL planning in **large** teams from the online survey with experts from academia.

method, the planning is performed on a more formal level, e.g., in the case of the financial company in terms of "target" feature model versions.

RQ2.3 *How suitable are feature-model evolution plans to plan SPL evolution?*

Semi-Structured Interviews with Industry Partners Our interview partners agreed that feature models are a very suitable artifact to plan the evolution of a product line. They also agreed that using feature models as main communication and planning artifact would improve management as well as development. However, the interview partners from the *automotive company* and the *financial company* argued that this also requires a feature-oriented development process for all stages of development. Moreover, they argue that management would have to use and understand feature models and, while this is desirable, they think that at the point in time of the interview, this is not realistic.

Online Survey with Experts from Academia In our online survey, we tried to distinguish between suitability for small and for large development teams. The sentiment was that we expected planning with feature models for small teams to be unnecessary overhead as they typically communicate a lot and, thus, do not need to plan a long time ahead. In contrast, we expected that for large teams, the participants of the survey would conclude that feature models are suitable as they simplify the communication and provide a formal basis. We asked the participants to estimate the suitability of feature-model evolution plans to plan SPL evolution on a scale from 0 (unsuitable) to 5 (very suitably) for small and large development teams. Figure 4.11 and Figure 4.12 show the results of these two questions. Most of the survey participants agree that feature-model evolution plans are suitable for small teams.

For large teams, the answers are very diverse and even no participant assessed the suitability as very suitable (in contrast to small teams), which strongly deviates from our expectations. However, in the open text comments, multiple participants argued that feature modeling with large teams might lead to concurrent modification issues. Thus, we suspect that we mislead part of the participants with our differentiation between small and large teams.

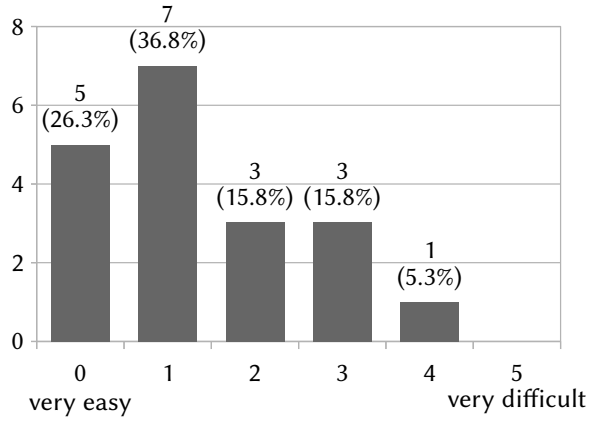


Figure 4.13.: Perceived difficulty to detect evolution paradoxes in a feature-model evolution example with **three** evolution steps from the online survey with experts from academia.

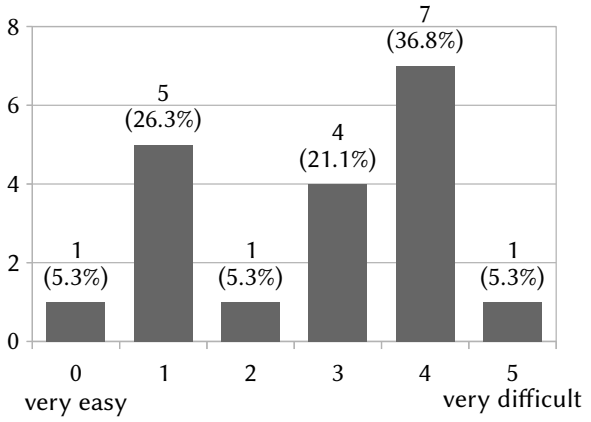


Figure 4.14.: Perceived difficulty to detect evolution paradoxes in a feature-model evolution example with **six** evolution steps from the online survey with experts from academia.

Especially the results from the interview partners show that our method to plan SPL evolution is highly desirable and could improve planning and communication in entire projects. However, this requires a feature-oriented development approach at the core of a project, i.e., also on management level. At the point in time of the interviews, this was not the case in the companies of our industry partners. It also seems that the experts from academia see high potential for feature-model evolution plans. However, the answers indicate that we mislead the participants of the survey and, thus, the conclusions we can draw from the respective questions are limited. In summary, we can answer **RQ2.3** positively, even if we mislead some participants by our confusing formulation of the question.

RQ2.4 *Do evolution paradoxes in feature-model evolution plans occur, and are they problematic?*

Semi-Structured Interviews with Industry Partners To answer **RQ2.4**, we presented and explained a feature-model evolution planning scenario that contained an evolution paradox in each interview. Subsequently, we asked our interview partners whether they experienced similar situations. In any case, we additionally asked whether evolution paradoxes are problematic, if they had to deal with real evolution paradoxes, and whether they lead to problems. All of our industry partners agreed that it is crucial to prevent evolution paradoxes if evolution is planned using feature models. They concluded that otherwise, evolution paradoxes may lead to severe problems that are potentially detected at a later point in time and require very expensive replanning. Especially if third-party developers are involved, such as suppliers in the *automotive domain*, replanning should be avoided. Furthermore, the industry partner from the *financial company* stated that they already encountered similar problems with their "target" and "current" feature models that diverged. The resulting manual fixing of the problems was very time-consuming and, thus, expensive.

Online Survey with Experts from Academia In the online survey, we asked the participants to rate the difficulty on a scale from 0 (very easy) to 5 (very difficult) to detect evolution paradoxes in two given evolution scenarios. For one small scenario with three evolution

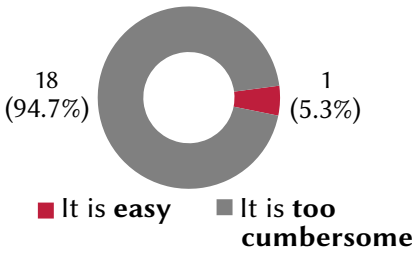


Figure 4.15.: Estimated feasibility to manually detect evolution paradoxes large-scale in feature models with many evolution steps.

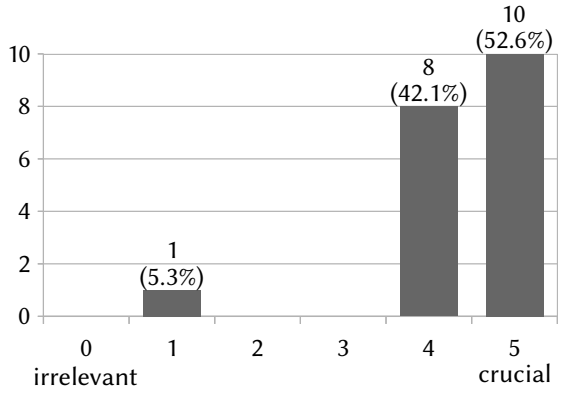


Figure 4.16.: Estimated value of a method for automatic detection of evolution paradoxes.

steps and for one scenario with six evolution steps. Figure 4.13 shows the results for the scenario with three evolution steps. Figure 4.14 shows the results for the scenario with six evolution steps. While the participants think that detecting evolution paradoxes is rather easy in the example scenario with three evolution steps, the estimated difficulty strongly increases for the example with six evolution steps.

Our results indicate that evolution paradoxes are a real problem and that they can lead to severe problems. Additionally, detecting and fixing them is a complex task, even for experts for the comparably small example scenario with six evolution steps. Consequently, we conclude that preventing the introduction of evolution paradoxes is crucial for a feature-model evolution planning method. Thus, we can answer for **RQ2.4** that evolution paradoxes occur also in real-world feature models and that they are problematic.

RQ2.5 *How valuable is an automated mechanism to detect evolution paradoxes?*

Semi-Structured Interviews with Industry Partners We directly asked this question to our interview partners from industry. They all conclude that an automatic detection mechanism for evolution paradoxes is very valuable if not essential when planning with feature models. The interview partners from the *automotive industry* and from the *web-application company* think that manual detection is not feasible for real-world feature-model evolution plans. In the *financial service company*, the "target" feature model and the "current" feature model serve similar purposes as our feature-model evolution plans. As stated in the answers for **RQ2.4**, they run into problems similar to evolution paradoxes, and detecting and fixing them is time-consuming and expensive manual labor. Moreover, they stated that this would be a very valuable and one of the most needed functionalities in their feature-modeling IDE.

Online Survey with Experts from Academia In the online survey, we asked the participants to assess the feasibility to manually detect evolution paradoxes in large-scale feature models with hundreds of features and many evolution steps. Additionally, we asked them to estimate the value of an automatic evolution paradox detection method from 0 (irrelevant) to

5 (crucial). Figure 4.15 shows the distribution of selections regarding the feasibility of manual detection for large-scale feature models. The results show that nearly all participants (**18 out of 19**) concluded that it is too cumbersome. Unfortunately, we did not receive an explanation of the one participant that selected that it would be easy. Figure 4.16 shows the estimation of the participants regarding the value of an automated evolution paradox detection mechanism. More than half (**52.6%**) selected that is even crucial to have such a mechanism. Again **one** participant selected 1, i.e., that is not very important to have such a mechanism, but without any further explanation.

In summary, the data shows that nearly all asked experts agree that an automatic evolution paradox mechanism is very important if not essential (**RQ2.5**).

Threats to Validity

Our empirical assessment is subject to threats to validity. We performed three interviews with industry partners and sent an online survey to experts from academia. The results may be biased as we only interviewed three industry partners and, thus, representativity may not be given. Additionally, our interview partners are engineers that are familiar with feature models and frequently work with them. However, we argue that our results are a strong indicator as the interview partners are employed at companies from very different domains and the integration of feature model (evolution) in their development process is very diverse. Nonetheless, we did not interview employees that work in management positions. Thus, our results are characterized by a very technical view.

In the interviews as well as in the online survey, we might have posed suggestive questions. We tried to reduced this threat by posing many open questions and, in the interviews, letting the interview partners elaborate on their problems before introducing evolution paradoxes.

The evaluation of our online survey might be subject to threats to validity as we only asked experts from the feature-modeling research domain. As feature modeling is their own field of research, estimating the value of feature models as evolution planning artifacts as high seems natural. However, even if these researchers frequently deal with feature models and their evolution, they assessed evolution paradox detection as complicated and automated mechanisms as crucial. Thus, in combination with the interviews with our industry partners who claim that feature-model evolution planning is beneficial, we argue that our method in its entirety is important.

4.5. Related Work

Ample research has been conducted in the field of model inconsistencies in general. To the best of our knowledge, no research exists that considers the evolution planning of models and resulting challenges regarding inconsistencies. In this section, we present the most relevant existing research for our topic. First, we give a brief overview of research dealing with inconsistencies in models in general. Second, we elaborate on work that explicitly addresses inconsistencies of SPL artifacts. Finally, we discuss work that considers feature models in particular.

Generic Model Inconsistencies General-purpose constraint languages have been developed that enable the definition of model consistency rules. Examples of such languages are the OCL[2] and Xpand Check language[3]. Constraints in such languages can be typically defined for arbitrary

[2] Object Management Group OMG. Object Constraint Language, Version 2.4 formal/2014-02-01, February 2014
[3] http://www.eclipse.org/modeling/m2t/?project=xpand

models and can be verified automatically. However, as the languages are very generic, particular situations such as evolution cannot be covered by those languages. For instance, defining constraints that ensure the consistency of a TFM at each evolution step would be very complicated, if not impossible. Additionally, verifying those constraints in the presence of evolution is typically very inefficient. Thus, these approaches are not specific to feature models or SPLs, but try to provide a solution in a more general manner.

Groher et al. [GRE10] consider the evolution of constraints and how they affect the consistency of arbitrary models. In particular, they identify model elements that are affected by changed consistency constraints by observing the behavior of consistency rules when being evaluated. Thus, when re-verifying changed constraints it is not necessary to analyze entire models, but only relevant parts. In our scenario, the constraints remain stable, as the well-formedness rules of TFMs do not change. In contrast, our models change and need to be re-verified. As we analyze inconsistencies based on evolution operations formalized in execution semantics, we only verify model elements that are affected by a change.

Kehrer et al. [KKT13] consider challenges when patching and merging model versions. In particular, these activities may lead to inconsistent models as model merging often only works on a textual basis. To provide a remedy, they introduce consistency preserving edit scripts that consist of script operations transforming one model version to another while preserving the model's consistency after each operation. However, they do not consider the introduction of intermediate evolution steps.

Software Product Line Artifact Inconsistencies Some researchers investigated how to ensure consistency of SPL artifacts in general. Czarnecki and Pietroszek provide a method of OCL constraints for feature-based model templates [CP06]. These model templates are metamodels with feature-based annotations. The annotations define feature configurations for which certain metamodel elements are available. However, certain features configurations may lead to inconsistent resulting metamodels, e.g., if the metamodel contains a dangling reference. The authors provide an analysis to automatically ensure that only consistent metamodels can be generated from a feature-based model template. Principally, Czarnecki and Pietroszek consider variability in space, i.e., feature selections lead to modified (meta) models [CP06]. Technically, this is not very different from variability in time, i.e., the evolution of (meta) models. Thus, their technique could be used to annotate a model with evolution steps instead of features and the resulting version could be checked for consistency. However, they do not consider intermediate evolution steps and the analysis would iterate over each version which is very inefficient.

Vierhauser et al. [VGH+12] provide a sophisticated generic framework to verify the consistency of heterogeneous product-line artifacts. In particular, their framework enables to specify consistency constraints for arbitrary artifacts which can also be used to ensure consistency *between* artifacts. An extensible *artifact facade* enables to handle arbitrary product-line artifacts. The *consistency framework* then instantiates the constraints for each artifact they apply to. In a *scope database*, they track verified and changed artifacts which enables incremental re-evaluation upon artifact modification. Thus, after evolution, only the constraints are verified that are relevant for the changed artifacts. However, they do not have a notion of evolution planning and can just verify the most recent artifact version. Thus, this method is not directly applicable to verify feature-model evolution plans with intermediate evolution steps.

Feature-Model Inconsistencies Multiple researchers addressed the topic of feature-model inconsistencies in various facets. Arcaini et al. consider the case if the constraints imposed by the feature model allow to derive valid configurations, but in fact undesired [AGV17]. They provide an automatic repair mechanism that fixes the feature model to prohibit the undesired configurations. To this end, new constraints are created, existing constraints are removed or modified. However, they do not consider structural inconsistencies of feature models nor evolution.

Quinton et al. investigate how the evolution of cardinality-based feature models may lead to range inconsistencies [QPB+14]. Cardinalities in feature models enable to have multiple instances of a feature in a configuration and they enable to specify lower and upper bounds of selected features in a group. However, the evolution of a feature model may lead to inconsistencies which may lead to unselectable features, features that must be selected despite being optional, cardinality values that can never be achieved in a configuration, or even to feature models that do not have any valid configuration. For instance, a newly introduced feature may require five instances of a second feature but the cardinality of the second feature's group only allows up to four feature instances. Quinton et al. do not consider structural inconsistencies apart from cardinality ranges and do not consider intermediate evolution steps [QPB+14]. Nonetheless, a combination of their and our methods could be beneficial when considering cardinality-based feature models which our method does not address.

Guo et al. define consistency rules for feature models that cover structural and semantical consistency [GWT+12]. The structural consistency rules define, inter alia, that a feature model must be a directed acyclic graph. Thus, removing a feature would lead to its children being unconnected which would be an inconsistency. The semantical inconsistency rules define that each feature must be selectable in at least one configuration, each optional feature must be deselectable in at least one configuration, and at least one valid configuration exists. In the next step, they identified which feature-model evolution operation may violate which consistency rule. They use this information to define *evolution strategies* which can be applied to fix an inconsistency that an evolution operation introduced. For instance, they define an evolution strategy to remove child features of a removed feature to preserve the property of a graph. They provide a formalization for feature models and consistency rules. However, it remains unclear how they verify these rules and how they implemented it. Moreover, it is unclear whether they incorporate their insights about consistency-breaking evolution operations in their verification. In contrast to our work, they do not consider feature-model evolution plans nor intermediate evolution steps. Our method goes beyond and verifies the consistency of all the following evolution steps after an operation. We do not provide a concept similar to evolution strategies as this is significantly more complex for planned feature-model evolution. It is not possible to provide appropriate solutions to fix inconsistencies introduced in an intermediate evolution step without additional domain knowledge. For instance, if a feature is deleted, but in a future step a child feature is already planned to be added to that first feature, it is unclear whether the feature should not be deleted at all, whether the child features should not be added, or whether the child features should be added to another parent feature in the feature tree. This illustrates the complexity of potential evolution strategies in the presence of intermediate evolution steps. Nonetheless, this is an interesting research direction for future work.

4.6. Chapter Summary

In this chapter, we addressed **Challenge 2: Consistent Feature-Model Evolution Replanning** by providing a concept to plan feature-model evolution while being able to replan without introducing inconsistencies. We addressed this challenge by answering the research questions **RQ2.1 – RQ2.5** that contribute to answering the first part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**. In a first step, we formalized well-formedness rules for feature models and TFMs. Then, we analyzed which types of inconsistencies, aka *evolution paradoxes*, exist and which evolution operation can cause them. Finally, we formalized the evolution operations and the well-formedness rules in execution semantics that enable us to detect evolution paradoxes in feature-model evolution plans.

We provide a fully-featured tool implementation integrated into DarwinSPL that enables users to plan and replan feature-model evolution while preventing the introduction of evolution paradoxes. In our evaluation, we have empirically shown that feature-model evolution planning and replanning is relevant in both academia and industry. Additionally, the industry reported that evolution paradoxes are a real problem and that automatic detection would be very valuable. Finally, we have shown the scalability of our method by applying it to existing feature-model evolution scenarios.

In summary, we present a method that makes use of TFMs presented in Chapter 3 as a basis to plan feature-model evolution. While TFMs conceptually enable replanning, evolution paradoxes can be easily introduced which we address with the contribution of this chapter. Thus, the first step to perform and plan SPL evolution is to adapt the feature model. We are now able to ensure structural consistency of feature models when replanning this evolution which answers the first part of **RQ2 – Feature-Model Inconsistency and Anomaly Prevention** and meets **Challenge 2: Consistent Feature-Model Evolution Replanning**. Another open challenge is that *semantical* consistency may be violated by performed evolution operations. For instance, a certain feature cannot be selected anymore. Such semantical inconsistencies in feature models are denoted in the literature as *feature-model anomalies* [BSR10]. In the next chapter, we will address how to detect these anomalies by incorporating the entire evolution timeline of a feature model. In combination with this chapter, we will answer **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention** in its entirety.

The research we present in this chapter raises several opportunities for future work. First and most importantly, we are currently not able to determine which intermediate operation caused an evolution paradox if multiple intermediate operations are analyzed at the same time. As a workaround in DarwinSPL, the entire plan is analyzed each time an intermediate operation is executed and, thus, a paradox-causing evolution operation can be identified. However, with the execution semantics, we only find the last operation which in fact results in an inconsistent feature model. However, this operation might be existent before introducing the intermediate operation and did not cause any inconsistency until then. As the intermediate operation changes the basis for the operations that are scheduled for subsequent time points, we only detect that the preconditions in the semantics rules for the subsequent operations are not met. Thus, for future work, it is sensible to analyze which intermediate operation resulted in that changed basis for subsequently scheduled evolution operations that would create an inconsistency.

The execution semantics we present detect only the first evolution paradox, then stop their execution, and report an error. This limits our method to detect only one evolution paradox at a

time. As a workaround in DarwinSPL, we currently analyze each intermediate evolution operation directly after its creation. Thus, we do not need to find more than one evolution paradox as we would prohibit the creation of paradoxical evolution operations. Thus, for future work, it would be sensible to devise an execution semantics that is able to proceed to execute the operations which requires appropriate means for error recovery.

5 Detecting and Explaining Anomalies in Feature-Model Evolution

The contents of this chapter are largely based on the work published in [NMS+18, MNS+18, MNS+17, SSK+20].

Summary *When devising feature models, design flaws, so-called anomalies, can occur. To fix such anomalies, developers need to understand their cause. However, for large evolution timelines and large feature models, anomaly explanations provided by existing methods may become very long and, as a consequence, hard to understand. In this chapter, we provide a method for anomaly detection that, by encoding the entire feature-model evolution timeline in one solver request, identifies all anomalies present in a feature-model evolution timeline. As understanding the cause for an anomaly is crucial for fixing it, our method identifies the evolution step of anomaly introduction and explains which of the performed evolution operations led to it. Using this information, we are able to provide more expressive anomaly explanations while reducing explanation complexity by focusing on the identified evolution operations. In our evaluation, we verify that our method correctly identifies all anomalies in an entire feature-model evolution timeline and correctly explains them using the causing evolution operations. Furthermore, we show that our method significantly reduces the complexity of generated explanations.*

Devising an SPL is a very complex and challenging task. This also concerns feature models. As a consequence, design flaws may be introduced. Similar to code smells for single software systems [Fow99], *anomalies* in feature models indicate bad modeling which may lead to errors [BSR10]. For instance, a *dead feature* anomaly exists if a feature can never be selected in a valid configuration whereas a *void feature-model* anomaly implies that no valid configuration at all can be derived from a feature model. While some anomalies may be deliberately introduced or tolerated, it is typically desirable to fix them. However, fixing anomalies is a complex task and, thus, entails significant costs [KAT16].

Numerous approaches exist to detect anomalies [Hem08b, ML04]. Other researchers support fixing anomalies by providing explanations for them [KAT16, MTS+17, Bat05, RGM+14, EPH09, FBG+13, KSR13, LSW15, Tri12, TBR+06]. However, for large feature models, explanations can have a significant size as a large percentage of features and cross-tree constraints contribute to the corresponding anomaly. This makes it very complex to understand the explanation. Thus, fixing the anomaly is time-intensive despite support. For instance, a recent bug in the tool FeatureIDE [MTS+17] resulted in few group types of feature models to be changed.[1] In consequence,

[1] https://github.com/FeatureIDE/FeatureIDE/issues/662

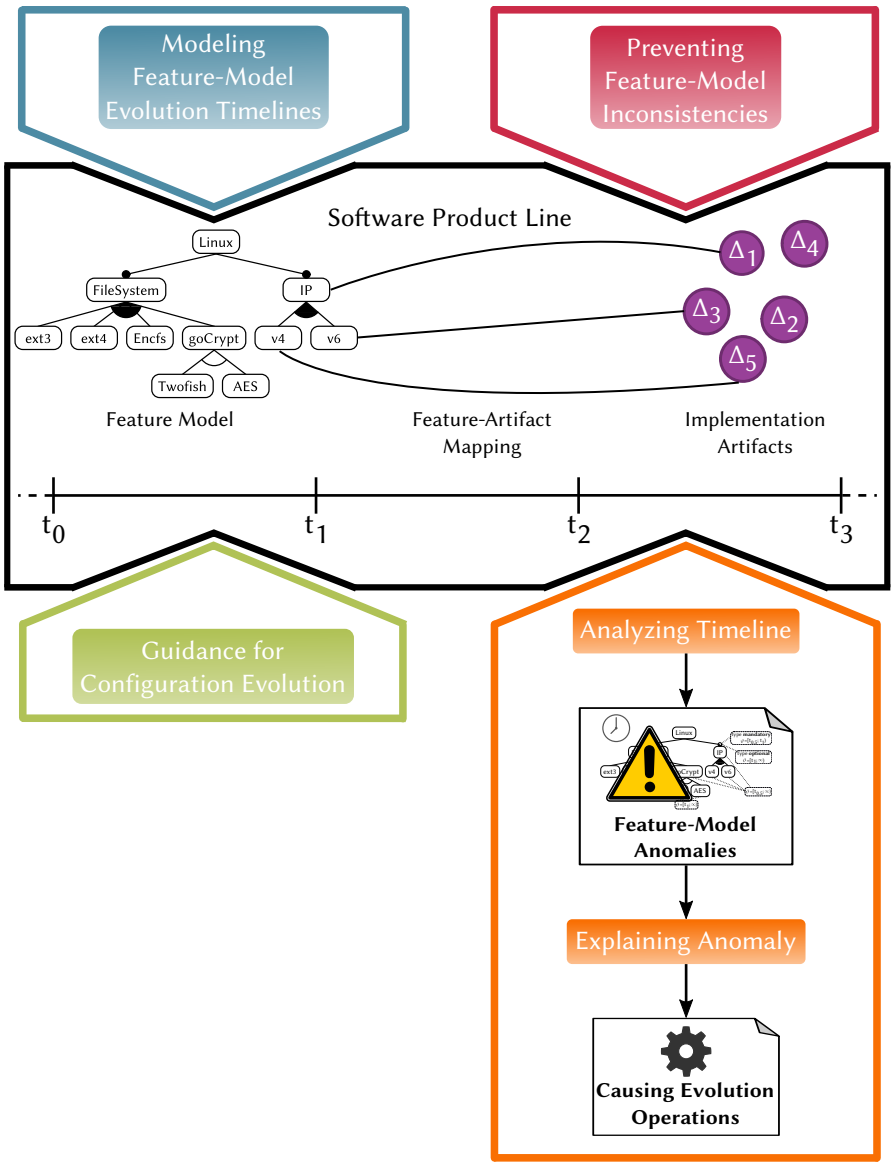


Figure 5.1.: Contribution overview – Step 3, Detecting and Explaining Evolution Anomalies.

OR groups were interpreted as AND groups with only MANDATORY features if any of the group's child features has own child features. For a large feature model from industry (712 features and 1141 cross-tree constraints), this resulted in three changed group types making the feature model *void*. The explanation of this anomaly contained 91 cross-tree constraints and 92 features were involved. Thus, engineers had to inspect all these cross-tree constraints and associated features despite the immediate cause being the change of three group types.

As evolution yields additional complexity for SPLs as particularly long-living systems, the likelihood that engineers inadvertently introduce anomalies increases. Moreover, planning and replanning feature-model evolution may result in many evolution steps with many evolution operations. Consequently, anomalies are not just introduced in the most recent feature-model version but may be distributed throughout the entire timeline. Without proper detection methods, introduced anomalies remain undetected and may cause errors. Additionally, fixing an anomaly is even more

challenging if no further information on the causing evolution operations is provided as the above-mentioned large explanation illustrates. However, existing approaches do not incorporate feature-model evolution [Hem08b, ML04, KAT16, MTS+17, Bat05, RGM+14, EPH09, FBG+13, KSR13, LSW15, Tri12, TBR+06]. Thus, we need a method to detect anomalies in an evolution timeline and to provide explanations that help engineers to understand the cause for the anomaly introduction which is addressed by **Challenge 3: Detecting and Explaining Anomalies in Feature-Model Timelines**.

In this chapter, we meet this challenge by incorporating feature-model evolution information of a TFM to detect and explain anomalies introduced during evolution. This answers the second part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**. Figure 5.1 shows the contributions of this thesis with more details on detecting and explaining anomalies. After modeling the entire feature-model evolution timeline in a TFM (cf. Chapter 3), we ensured consistency of this evolution (cf. Chapter 4). The contributions of this chapter are used to fix anomalies of a TFM by supporting anomaly detection and explanation. Thus, in combination with the contributions of Chapter 4, we can answer **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention** in its entirety.

In particular, we make the following contributions in this chapter. In Section 5.1, we propose a method that detects anomalies in the evolution timeline and pinpoints the evolution step of anomaly introduction by analyzing a TFM in its entirety (as opposed to evolution steps individually). In Section 5.2, we introduce a novel concept for explaining anomalies by identifying causing evolution operations and, thus, reducing explanation complexity. We evaluate our method in Section 5.3 by presenting our implementation and tool support, by qualitatively evaluating our method, by measuring performance in a qualitative evaluation, and by measuring explanation reduction rates. We discuss research that is related to ours in Section 5.4. In Section 5.5, we close this chapter by providing a summary and an outlook on future research directions for further improving anomaly detection and explanation.

5.1. Detecting Anomalies in Feature-Model Evolution Timelines

Feature-model anomalies are a well-studied topic. Benavides et al. [BSR10] provide an overview of the most common anomalies considered in the literature. Among others, the following, most relevant, anomalies are discussed:

- A *void feature model anomaly* exists if no valid configuration can be derived from a feature model (i.e., the feature model is void).
- *Dead features* cannot be selected in any valid configuration. If undetected, this anomaly leads to unnecessary implementation effort for the dead features or to an unintentionally reduced configuration space.
- A *false-optional feature* is modeled as OPTIONAL, but is part of every valid configuration in which its parent feature is selected. Thus, this feature is effectively MANDATORY despite it has been devised as OPTIONAL, which leads to a potentially unintentionally reduced configuration space.

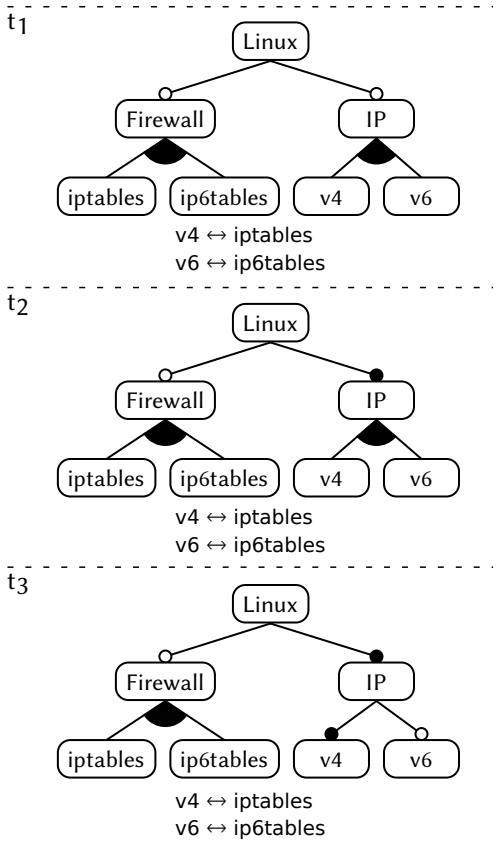


Figure 5.2.: Evolution of the IP feature of the Linux kernel running example feature model.

- *Redundant constraints* are cross-tree constraints that express the same restrictions as the tree structure of the feature model and/or other cross-tree constraints. This anomaly leads to reduced maintainability of a feature model as modifications have to be performed in multiple places if those constraints are involved.

Several approaches exist to detect [Hem08a, ML04] anomalies, e.g., using SAT solvers. Further research provides approaches to explain anomalies in terms of constraints that lead to an anomaly [KAT16, MTS+17, Bat05, RGM+14, EPH09, FBG+13, KSR13, LSW15, Tri12, TBR+06]. To the best of our knowledge, none of the existing methods is considering evolution. Thus, when analyzing a feature-model evolution timeline, the entire analysis has to be performed for each evolution step separately. Engineers then have to search manually for the evolution step in which an anomaly has been introduced, which can be hard if not unfeasible for large evolution timelines. This is even worse in concert with feature-model evolution planning and replanning, as in parallel, multiple evolution steps are planned far ahead, active development for the current feature-model version is performed, and intermediate changes are performed to replan. Even more important, when performing feature-model evolution, anomalies may be introduced by a small set of evolution operations, but the explanations using current methods may become extremely large (see the example in the introduction of this chapter with more than 90 involved constraints).

Running Example For instance, Figure 5.2 shows an extended evolution timeline of the Linux kernel feature model running example. This example focuses on the introduction of the `IP` feature

that has been planned for t_1. Two different versions of that protocol are provided in an OR group, namely `v4` and `v6`. Additionally, a `Firewall` feature is responsible for regulating access to the `IP` interface and has two specializations in an OR group, one for each IP version, i.e., `iptables` and `ip6tables` respectively. Two cross-tree constraints define that `v4` and `iptables` can only be selected together, and `v6` and `ip6tables` accordingly. In the initial feature-model version at t_1, the features are introduced as originally planned, i.e., `Firewall` and `IP` as OPTIONAL, and the sub features in OR groups. Shortly after the initial introduction, it becomes clear that no new system comes without network connectivity and, thus, the `IP` feature becomes mandatory at t_2. Moreover, the ongoing efforts to establish IP `v6` is delayed and, thus, engineers decide to make `v4` mandatory in t_3 to ensure that all systems can still communicate. However, the changes at t_2 and t_3 cause two anomalies. At t_2, `Firewall` becomes false-optional as either `v4` or `v6` must be selected and, consequently, `iptables` or `ip6tables` must be selected as well. Changing `v4` to mandatory at t_3 causes also `iptables` to become false-optional. While this small example is easy to handle, such changes in large real-world feature models with hundreds of features can result in anomalies that are hard to find, to understand, and to fix.

Constructing Satisfiability Problems When searching for anomalies, existing approaches typically construct satisfiability problems and solve them by querying off-the-shelf solvers. In these satisfiability problems, all features are translated into variables that can be either *true* (i.e., selected) or *false* (i.e., deselected). The constraints imposed by the feature-model structure and the cross-tree constraints are translated into a formula FM_c. A solution of that formula, i.e., variable assignment A that satisfies FM_c, then corresponds to a valid configuration and, thus, FM_c represents all possible valid configurations. For instance, Listing 5.1 shows the propositional formula for the feature model of the running example and its cross-tree constraints at t_1. Lines 1 – 6 represent the constraints imposed by the feature-model structure and lines 7 – 8 are the cross-tree constraints.

Listing 5.1: Propositional formula of the running example at t_1.

```
1  Linux ∧
2  ((Firewall ∨ IP) → Linux) ∧
3  ((iptables ∨ ip6tables) → Firewall) ∧
4  (Firewall → (iptables ∨ ip6tables)) ∧
5  ((v4 ∨ v6) → IP) ∧
6  (IP → (v4 ∨ v6)) ∧
7  (v4 ↔ iptables) ∧
8  (v6 ↔ ip6tables)
```

A void feature-model anomaly can be detected by trying to determine a solution for such a satisfiability problem. If no solution exists, the feature model is void. To search for other anomalies, additional constraints are added to that formula and checked for satisfiability. For instance, it can be checked whether a feature f is dead by trying to find a solution for the formula $FM \wedge f$. If no solution exists, f is dead. Similarly, f is false-optional if no solution for the formula $FM \wedge f_p \wedge \neg f$ exists, with f_p being the parent feature of f.

5.1.1. Encoding the Evolution Timeline

Existing approaches do not incorporate evolution information to detect or explain anomalies. As a consequence, each evolution step of a feature model has to be analyzed separately. In the presence of evolution planning and replanning, this may lead to late detection of anomalies. Thus, subsequently performed changes may base on changes that introduced an anomaly. This is a similar problem as for evolution paradoxes where changes may rely on inconsistency introducing evolution operations that have to be reverted (cf. Chapter 4). Consequently, if an anomaly is fixed, all changes that relied on the change that introduced an anomaly may have to be revised if not reverted. As a result, anomalies might not be fixed at all, which may lead to more severe problems at later points in time and to decay of feature-model quality.

We provide an evolution-aware anomaly detection that analyzes the entire evolution timeline of a TFM. For the evolution steps of our running example (cf. Figure 5.2), many formula parts remain the same. This results in redundancies in solver queries if multiple evolution steps are analyzed. The idea of the evolution-aware anomaly detection is to incorporate feature-model evolution for anomaly detection and explanation by encoding the entire evolution timeline in one set of variables and formula to give to a solver. This way, we are able to i) reuse the solver for parts of the formula that remain stable over multiple evolution steps such that the entire evolution timeline is analyzed automatically; ii) detect the evolution step at which an anomaly first arose; iii) explain the anomalies with evolution operations performed by engineers. In the following, we describe how we encode the evolution timeline as one formula. While we focus on void feature-model, dead feature, and false-optional feature anomalies, our technique can be used for other anomaly analyses as well.

We utilize the information on the evolution provided by a TFM. We encode the notion of evolution into one single request. For this purpose, we *tag* evolving parts of the formula with the time intervals for which they are temporally valid. Parts that remain the same for the entire evolution timeline can be left as they are, i.e., without any tagging. These tags tell the solver for which point in time it needs to consider the tagged parts. In particular, we introduce a new *evolution* variable, representing the evolution steps, that we use for the tagging. As we may have multiple evolution steps, this variable has to have a larger domain than the variables representing features that can just have *true* and *false* as values. Thus, we decided to use a formula in a first-order style notation which enables us to also use variables having an integer domain. A formula that is valid in the interval $[t_i, t_j)$ is tagged with the evolution variable as follows:

$$(\textbf{evolution} \geq t_i \ \wedge \textbf{evolution} < t_j) \rightarrow (\textit{original formula})$$

As the evolution variable represents all evolution steps, we need to identify all relevant steps and set the variable domain accordingly. For this purpose, we make use of the temporal elements of TFMs. We identify all relevant steps by inspecting the temporal validities ϑ_e of all temporal elements $e \in E$ of a TFM. In particular, we determine the set $steps = \{\vartheta_{since}, \vartheta_{until} | \vartheta_e = [\vartheta_{since}; \vartheta_{until}), e \in E\}$. The domain of the *evolution* variable is $[0; n]$, where $n = |steps| - 1$ is the number of unique evolution steps identified in the TFM. For instance, in the running example, the respective set is determined as $steps = \{t_1, t_2, t_3\}$, i.e., $n = 2$. The value 0 for the evolution variable represents the state before the first evolution step (e.g., the TFM at t_1 of the running example).

Table 5.1.: Encoding rules to clauses of a propositional formula and linking of TFM elements.

TFM Property	Linked TFM Elements	Clause in Formula
Feature $\boldsymbol{R}$ is root Feature	Feature	R
Feature $\boldsymbol{P}$ is the parent feature of features $\boldsymbol{F_1}, \ldots, \boldsymbol{F_i}$	Feature Group Composition Parent-Feature-Child-Group Relation	$(F_1 \vee \cdots \vee F_i) \rightarrow P$
Features $\boldsymbol{F_1}, \ldots, \boldsymbol{F_i}$ are mandatory sub features of feature $\boldsymbol{P}$	Feature Type	$P \rightarrow (F_1 \wedge \cdots \wedge F_i)$
Features $\boldsymbol{F_1}, \ldots, \boldsymbol{F_i}$ are in an or group with parent feature $\boldsymbol{P}$	Group Type	$P \rightarrow (F_1 \vee \cdots \vee F_i)$
Features $\boldsymbol{F_1}, \ldots, \boldsymbol{F_i}$ are in an alternative group with parent feature $\boldsymbol{P}$	Group Type	$F_1 \leftrightarrow (\neg F_2 \wedge \cdots \wedge \neg F_i \wedge P) \wedge$ $F_2 \leftrightarrow (\neg F_1 \wedge \neg F_3 \wedge \cdots \wedge \neg F_i \wedge P) \wedge$ ... $F_i \leftrightarrow (\neg F_1 \wedge \cdots \wedge \neg F_{i-1} \wedge P)$
Features $\boldsymbol{F_1}, \ldots, \boldsymbol{F_i}$ whose $\boldsymbol{\vartheta_{since}}$ is larger than $\boldsymbol{t}$	Feature	$(evolution < t) \rightarrow (\neg F_1 \wedge \cdots \wedge \neg F_i)$
Features $\boldsymbol{F_1}, \ldots, \boldsymbol{F_i}$ whose $\boldsymbol{\vartheta_{until}}$ is less or equal than $\boldsymbol{t}$	Feature	$(evolution \geq t) \rightarrow (\neg F_1 \wedge \cdots \wedge \neg F_i)$

Listing 5.2: Formula tagged with the evolution variable that encodes the entire evolution timeline of the running example.

```
1  Linux ∧
2  ((Firewall ∨ IP) → Linux) ∧
3  ((evolution ≥ 1) → (Linux → IP)) ∧
4  ((iptables ∨ ip6tables) → Firewall) ∧
5  (Firewall → (iptables ∨ ip6tables)) ∧
6  ((v4 ∨ v6) → IP) ∧
7  ((evolution < 2) → (IP → (v4 ∨ v6))) ∧
8  ((evolution ≥ 2) → (IP → v4)) ∧
9  (v4 ↔ iptables) ∧
10 (v6 ↔ ip6tables)
```

Each clause of a formula TFM_c is generated by a certain TFM element. Table 5.1 shows how elements of a TFM are encoded. Our encoding is based on the encoding provided by Mendonca et al. [MWC09]. The first column describes properties of a TFM that are encoded, the second column shows which actual elements of a TFM are used to derive the clauses, and the third column shows the resulting clause in the formula TFM_c. For instance, Listing 5.2 shows the formula with tagged parts for the entire evolution timeline of the running example. Line 2 is generated as the features `Firewall` and `IP` are part of a child group of the feature `Linux`. As each TFM element is attributed with a temporal validity, i.e., the time interval $[\vartheta_{since}; \vartheta_{until})$ at which the element is temporally valid, we can directly derive the tags from the temporal validities. In the tagged formula,

only three clauses need to be tagged with the evolution variable and the remaining clauses can be reused for each evolution step. For instance, the type of the feature `IP` changed at t_2 from OPTIONAL to MANDATORY. Consequently, the TFM element representing the new MANDATORY type has a temporal validity of $[t_2; \infty)$. The derived clause of this element (cf. row three in Table 5.1) is tagged with $evolution \geq 1$ in TFM_c. Listing 5.2 illustrates this in line 3. Line 7 is the encoding of features `v4` and `v6` being in an OR child group of feature `IP` until t_3 (i.e., $evolution < 2$) and line 8 encodes that feature `v4` became MANDATORY starting from t_3 (i.e., $evolution \geq 2$). As the remaining parts of feature model did not change during the feature model's evolution, they are the same for each evolution step.

As during evolution, features are added or removed, we also need to encode that features cannot be selected at all at time points they are not temporally valid at. To this end, we defined the last two encoding rules in Table 5.1. In particular, we define one clause for each point in time at which an evolution operation occurred. Thus, for the running example, this would result in three clauses. We derive the time points by analyzing all existing temporal validities. Each unique time point contained in the endpoints of the temporal validity is then used to derive such a clause. In each of those clauses, it is defined that all features that are *not* temporally valid at the considered point in time cannot by selected, i.e., a negation of that feature is defined. For instance, if a feature f is deleted at t_1, a clause $(evolution \geq t_1) \rightarrow (\neg f_1)$ would be generated.

Note that our encoding enables the seamless analysis of the entire feature-model evolution timeline which includes past evolution history, currently performed evolution, and pre-planned evolution steps. After this translation and tagging, the formula can be used as a basis to detect anomalies using off-the-shelf solvers.

5.1.2. Solver Queries for Feature-Model Evolution Timelines

In the following, we describe how we detect anomalies in the entire evolution timeline of a TFM. To this end, we denote the tagged formula that encodes the entire timeline as TFM_c with the evolution variable e and features of the TFM as literals, $f \in F$. Formulas that represent simple feature models only contain Boolean variables. To be able to represent all evolution steps, we define e as an integer variable. As a result, we cannot use standard SAT solvers anymore as they are only able to reason on formulas containing only Boolean variables. To compensate this, we need to use more expressive solvers, such as CSP or SMT solvers.

To detect void feature model anomalies for a given time point t_i, we check the satisfiability of the tagged formula by setting $e = t_i$, i.e., $((e = t_i) \wedge TFM_c)$. If this formula is satisfiable, a satisfying assignment for all $f \in F$ exists, i.e., a valid configuration exists for time point t_i exists. Conversely, if the formula is unsatisfiable, no satisfying assignment for all $f \in F$ exists, i.e., the feature model is void at t_i. To find all void feature model anomalies in the entire evolution timeline, a solver iterates over the value domain of the evolution variable and sets the value of e accordingly.

Reasoning about feature anomalies, i.e., dead and false-optional features, is more complex as each feature has to be analyzed individually. To check whether a feature $f \in F$ is dead at time t_i, we check whether $SAT(f \wedge (e = t_i) \wedge TFM_c)$. Similar to the void feature model anomaly, if this formula is satisfiable, a satisfying assignment containing f exists, i.e., the feature f is not dead. Conversely, if the above-defined formula is unsatisfiable, f is dead. To determine whether a feature f is false optional works similarly: while for dead features, we enforce the considered feature to be selected, we enforce the parent of the considered feature f_p to be selected but the feature itself to be

deselected. Thus, we search for a solution by checking $SAT(f_p \wedge \neg f \wedge (e = t_i) \wedge TFM_c)$. If we find a solution for that formula, f is not false-optional, but if no solution exists, f is false-optional at t_i.

As a dead MANDATORY feature results in a void feature model and only OPTIONAL features can be false-optional, we only consider OPTIONAL features for the respective analyses. However, a feature's type may change during evolution, we need to know which features to check at which evolution step. To this end, we introduce an ordered list *toCheck* of tuples of features and time points whereas each tuple $(f_j, t_i) \in toCheck$ represents that the feature f_j is OPTIONAL at time point t_i. As we want to determine all anomalies in the entire feature-model evolution timeline, we iterate over all possible $(f_j, t_i) \in toCheck$. First, we set the evolution variable to t_i to specify that we search for anomalies at t_i. Subsequently, we check whether f_j is dead by adding the respective formula. If f_j is not dead, we replace the formula to check whether it is dead by the formula to check whether it is false-optional. We repeat this for all features contained in the entries of *toCheck* for the same time point. Afterwards, we proceed to the next time point contained in *toCheck* and repeat the previously described procedure. To optimize this procedure, we order the tuples in *toCheck* by the time point of the tuples. Consequently, we minimize the number of changes of the evolution variable value.

As the entire evolution timeline is encoded in the tagged formula, a solver can reuse findings from already analyzed points in time to find dead or false-optional features or to prove voidness faster for other points in time. For this purpose, incremental solvers can be used that allow the addition and removal of constraints on-the-fly without restarting the search from scratch. Additional optimizations can be used to speed up the analyses as well. For instance, if a feature is proven to be dead, all of its child features must be dead as well. However, we do not consider this kind of optimizations as we want to focus on the encoding of the entire timeline in one formula and to analyze the impact of that encoding.

In summary, we are able to determine all anomalies that exist in a feature-model evolution timeline. Compared to existing approaches, we are able to analyze the entire timeline using one formula and iteratively setting the value for the evolution variable. Thus, we need to create only one solver request and can reuse the solver and its prior findings for subsequent evolution steps. Moreover, we automatically know for which evolution step an anomaly has been found. This is possible as each anomaly has been found by solving a formula with a specific value for the evolution variable representing an evolution step. Consequently, we identify the time point when an anomaly has been introduced, e.g., the first time point a specific feature is dead.

5.2. Explaining Anomalies using Feature-Model Evolution Operations

Detecting all anomalies in a feature-model evolution timeline and pinpointing the time point of their introduction is the first step to fix the anomalies. A second building block that is necessary for engineers to be able to fix an anomaly is to *understand* the cause for that anomaly. Several methods exist that explain an anomaly in terms of parts of unsatisfiable formula parts to support engineers in fixing that anomaly [KAT16, MTS+17, Bat05, Hem08a, RGM+14, EPH09, FBG+13, KSR13, LSW15, Tri12, TBR+06]. However, for large feature models, anomaly explanations can have a significant size if many features and cross-tree constraints are involved. When engineers try to fix such an anomaly using existing methods, in the worst case, they have to study the entire explana-

tion to identify the cause of the anomaly, even if only a few evolution operations may be the cause (see the example of this chapter's introduction).

To overcome those limitations, we explicitly incorporate information on the feature-model evolution in two ways: first, we explain anomalies for the time point in which they were introduced taking over the task of searching for the point in time of anomaly introduction; second, we identify the feature-model evolution operations that caused an anomaly. Consequently, we reduce the explanation length focusing on the relevant parts and are able to narrow down the search for fixes for the respective anomaly.

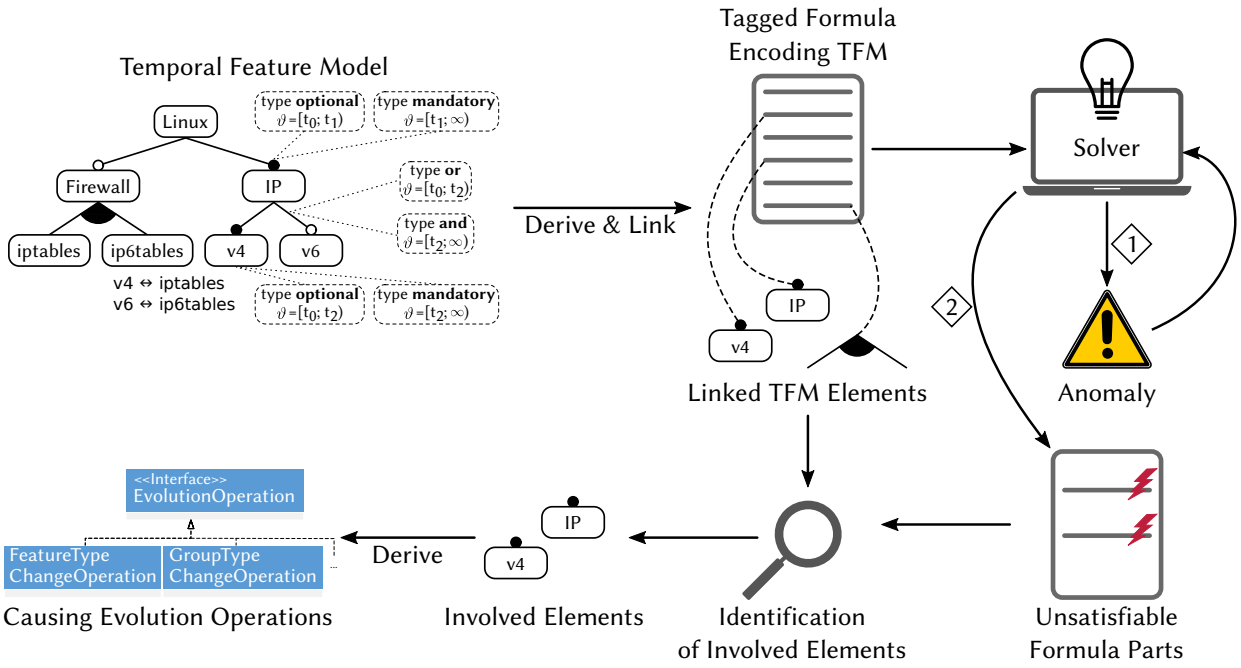


Figure 5.3.: Process that links TFM elements to tagged constraints, detects (①) and explains anomalies in terms of unsatisfiable formula clauses (②), identifies involved TFM elements using the linking, and identifies causing evolution operations applied to the identified TFM elements.

Typically, an anomaly is identified by showing that a certain formula cannot be satisfied (cf. Section 5.1). Existing approaches to compute anomaly explanations identify parts of this formula that cannot be satisfied and return them as an explanation. This is done in a second solver request that is specifically responsible for retrieving unsatisfiable formula parts. Evolution operations that caused an anomaly contribute to the existence of these formula parts. Thus, to identify the anomaly-causing evolution operations, we need to identify the evolution operations that have contributed to parts of the formula of the explanation.

Figure 5.3 shows the general process of the evolution-aware anomaly explanation. The first step is to derive the tagged formula as described in the previous section. While deriving each clause of the tagged formula, we link the TFM elements from which this formula part is derived. In particular, we link the formula parts to instances of the TFM metamodel classes (cf. Chapter 3, Figure 3.5) as described in Table 5.1. For instance, in Listing 5.2, the formula part in line 7 is derived from the MANDATORY feature type of the feature `v4` and, thus, we link this formula part to the respective instance of the class `TemporalFeatureType`.

In the second step, we detect and explain anomalies in three sub-steps. We iterate over the evolution variable and retrieve all anomalies using a solver ①. For each anomaly, we can query the same solver to explain the anomaly. We retrieve an explanation in the form of unsatisfiable clauses of the formula in ②. Subsequently, we combine the linking done in the first step with the unsatisfiable formula parts reported by the solver. We use this information to retrieve the TFM elements that are involved in the explanation of the anomaly.

In the last step, we derive the evolution operations affecting the identified involved elements. As elaborated in Section 4.4.1, we save the evolution operations that are performed to modify a TFM in a separate model. Thus, we scan this operation model and retrieve all operations affecting the elements involved in an anomaly. However, this may also include operations that have been performed in time points before or after the anomaly introduction. Thus, we categorize the identified evolution operations in *future* evolution operations, *involved past* evolution operations, and *causing* evolution operations. Causing evolution operations are all those operations that have been executed at the same time point at which the anomaly has been introduced. Thus, these operations immediately triggered the anomaly. Nonetheless, the involved past evolution operations may be helpful as additional information to understand how it came to an anomaly. *Future* evolution operations can be ignored for further computations.

Listing 5.3: Unsatisfiable formula clauses of the explanation for the false-optional feature anomaly of feature `iptables` of the running example.

```
1  Linux ∧
2  ((evolution ≥ 1) → (Linux → IP) ∧
3  ((evolution ≥ 2) → (IP → v4) ∧
4  v4 ↔ iptables
```

For instance, in the running example, the feature `iptables` becomes false-optional at t_3. Listing 5.3 shows the unsatisfiable formula parts that are returned by the solver. Among others, we retrieve the feature types of `IP` and `v4` as involved TFM elements. When scanning the operations model, we can identify two evolution operations that are affecting those two TFM elements: the feature type change of the feature `IP` from Optional to Mandatory at t_2 and the type change of the feature `v4` from Optional to Mandatory at t_3. As the feature `iptables` became false-optional at t_3, the type change of the feature `v4` is a causing evolution operation, whereas the type change of the feature `IP` is an involved past evolution operation. In summary, we reduced this exemplary anomaly explanation from four unsatisfiable formula parts to one causing evolution operation.

5.3. Evaluation

With the detection and explanation of anomalies in the entire evolution timeline of a feature model, we seek to answer the second part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**. To provide evidence that our anomaly detection answers this question, we evaluate our respective method by four means: first, we show feasibility by providing an implementation in the TFM modeling tool suite DarwinSPL including the detection of all anomalies in the timeline and the operation-based explanations; second, we qualitatively evaluate our method by verifying whether we correctly identify all evolution operations leading to an anomaly; third,

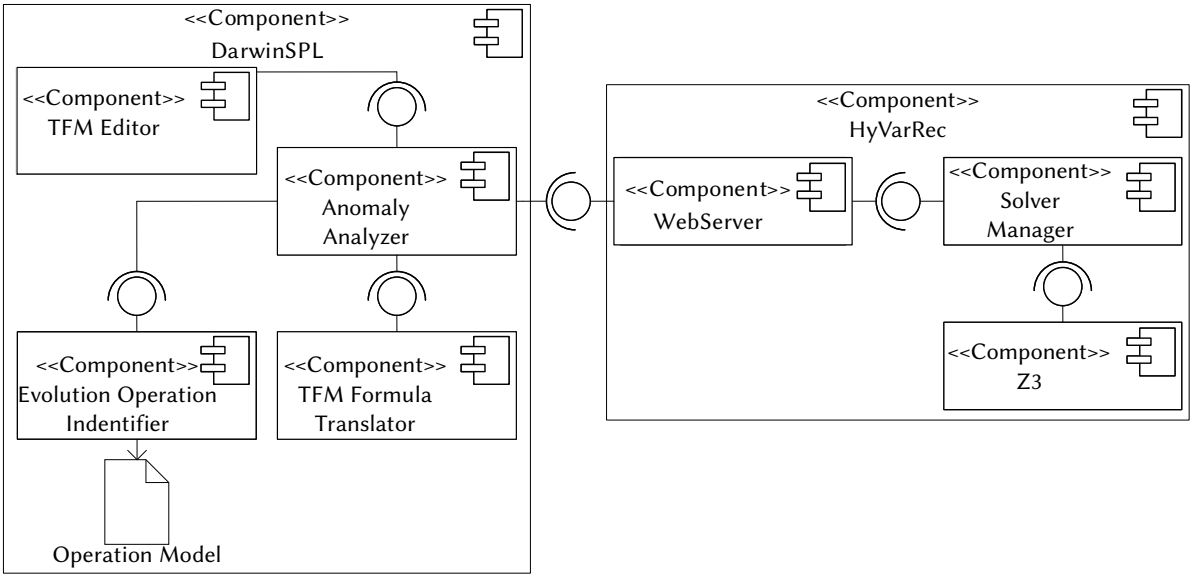


Figure 5.4.: Components involved in the evolution-aware anomaly detection and explanation.

we quantitatively evaluate whether our method scales to the evolution of real-world large-scale feature models; fourth, we quantitatively evaluate to what extent we are able to reduce anomaly explanation complexity compared to existing methods.

5.3.1. Implementation

We implemented our method in two tools: first, we implement all encoding, TFM element linking, evolution operation retrieval, and user interfaces in DarwinSPL; second, we implement a tool called HyVarRec[2] that is responsible for the solving part, i.e., iterating over the evolution variable, exchanging formula parts to identify the anomalies, and retrieving the unsatisfiable formula parts for anomaly explanations. Thus, HyVarRec can also be used to analyze other feature models (with evolution) than the TFMs of DarwinSPL.

As we described in Section 5.1, we encode the evolution stored in the TFM using the evolution variable. Consequently, we represent the evolution variable as an integer variable. However, standard SAT solvers only support propositional formulas with Boolean variables and, thus, they are not well suited to analyze the satisfiability requests with the encoded evolution timeline. In contrast, SMT solvers are able to reason about formulas containing integer variables. Thus, we use an SMT solver in HyVarRec to be able to reason on the evolution variable. In particular, HyVarRec is implemented as a RESTful webservice that uses the Z3 SMT solver [MB08] as backend.

Figure 5.4 shows the components that are involved in the anomaly detection and explanation. The detection is started using the anomaly view of the DarwinSPL editor. Figure 5.5 shows the TFM of the running example in DarwinSPL and the view of the detected anomalies. Then, the `Anomaly Analyzer` is instructed to start the anomaly detection. It uses the `TFM Formula Translator` which creates the tagged formula and additionally links the TFM elements to clauses. Subsequently, the tagged formula is sent to the `WebServer` of HyVarRec. The `Solver Manager` uses the tagged formula as input for the Z3 SMT solver and iter-

[2] `https://github.com/HyVar/hyvar-rec`

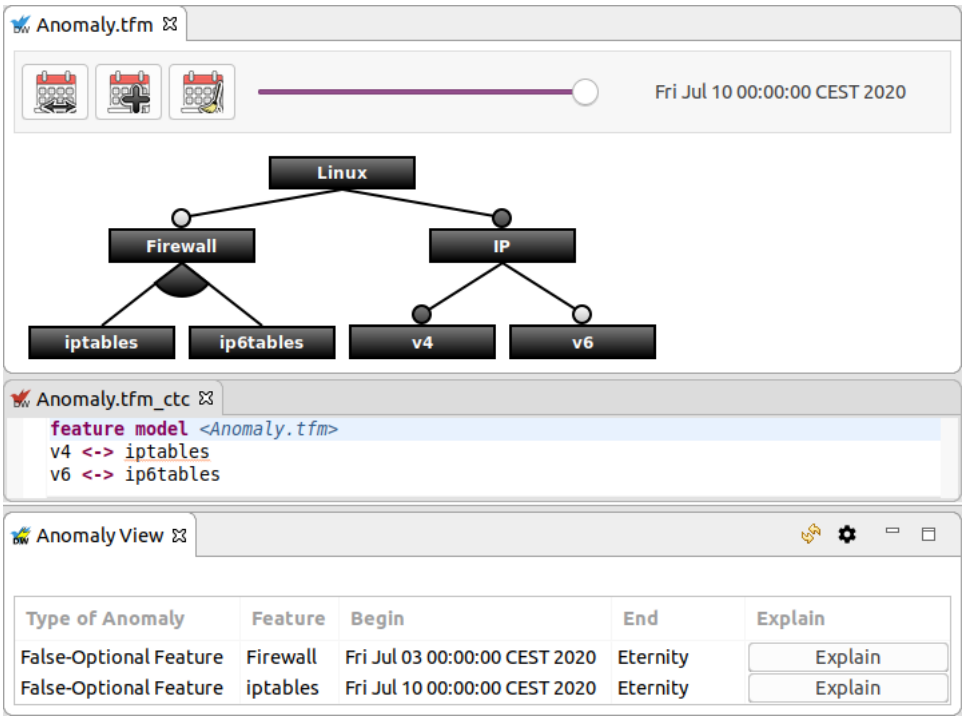


Figure 5.5.: Screenshot of detected anomalies in the running example of Figure 5.2 in DarwinSPL.

ates over the evolution variable. As a result, it sends back all detected anomalies with their respective time point of introduction. The anomaly view of the `TFM Editor` shows these anomalies including their type (i.e., void feature model, dead feature, false-optional feature), if applicable, the affected feature, and the time interval of the anomaly's existence. An additional button for each anomaly starts the respective anomaly explanation.

Upon pressing the explanation button, the TFM is translated again and the TFM elements are linked. A request to explain the respective anomaly is sent to the `WebServer` and the `Solver Manager` uses Z3 to retrieve the unsatisfiable core, i.e., the unsatisfiable clauses of the tagged formula. Since an anomaly may have multiple explanations, based on the internal search heuristics used for the SMT solver, we provide just one of them. However, this explanation is minimal in the sense that if one of the formula parts is removed, the formula becomes satisfiable.

HyVarRec sends the unsatisfiable clauses back as result and the `Anomaly Analyzer` forwards these and the TFM element linking to the `Evolution Operation Identifier`, asking to retrieve all involved evolution operations. The `Evolution Operation Identifier` collects and return the respective evolution operations from the `Operation Model`. The explanation is then presented by the `TFM Editor` to the engineers. Figure 5.6 shows the explanation and evolution operation for the false-optional feature anomaly of the feature `iptables` of the running example. To provide as much information as possible, DarwinSPL currently shows all unsatisfiable formula parts, all involved elements, and all involved evolution operations, i.e., also involved past evolution operations. However, engineers can focus on the causing evolution operations, such as the feature type change of the feature `v4` highlighted by a red frame in Figure 5.6.

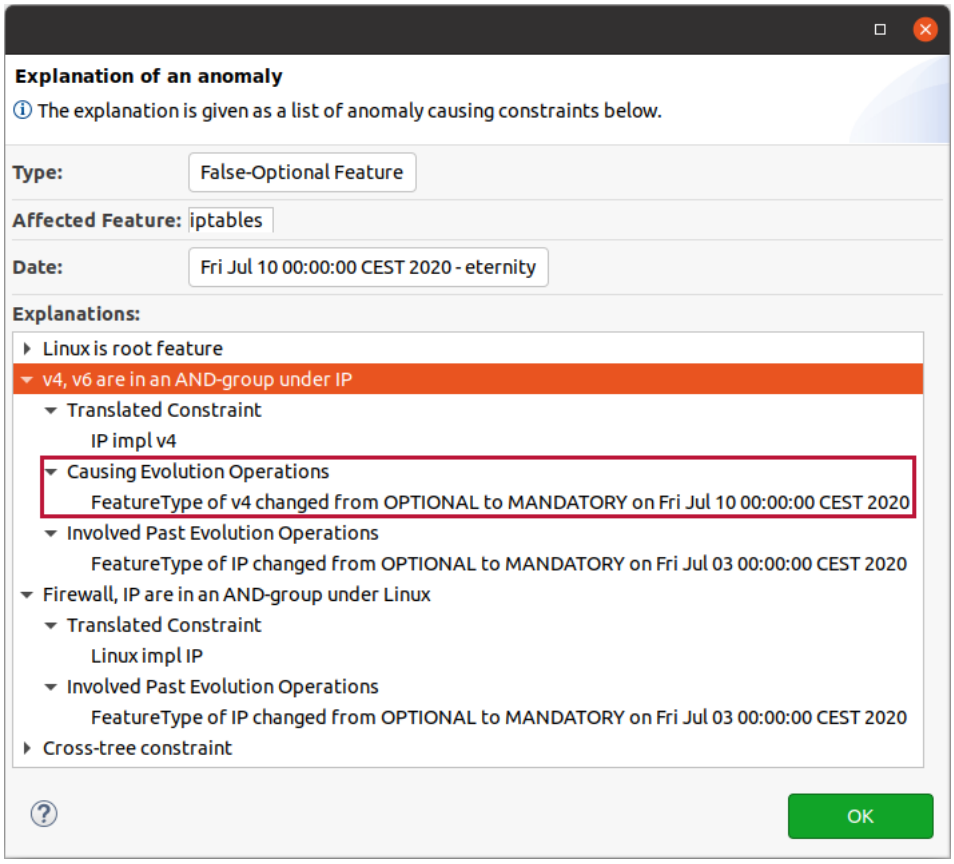


Figure 5.6.: Explanation for the false-optional feature anomaly of feature `iptables` of the running example in DarwinSPL.

5.3.2. Qualitative Evaluation

In the qualitative evaluation, we investigate whether our implementation works correctly which is a necessary prerequisite to answering the second part of **RQ2** positively. To this end, we pose two additional sub-research questions:

RQ2.6 Does our implementation identify all anomalies in a feature-model evolution timelines?

RQ2.7 Does our implementation correctly identify anomaly-causing evolution operations?

Setup We use two different types of subject systems to answer these research questions. First, we use two large-scale real-world feature models that already contain anomalies. Second, we use a medium-sized real-world feature model without any existing anomalies in which we manually seed anomalies to have a ground truth. All data related to the evolution operation description, the corresponding requests for HyVarRec, and the results can be found in our online repository.[3]

For the large-scale real-world feature models, we use the feature models of *Automotive02* and *FinancialServices01* which we already used in Chapters 3 and 4.[4] The evolution timeline of *Automotive02* contains four versions, between 14,010 (version 1) and 18,616 (version 4) features, and between 666

[3] `https://gitlab.com/evolutionexplanation/evolutionexplanation`
[4] `https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples/FeatureModels`

(version 1) and 1,369 (version 4) cross-tree constraints. The evolution timeline of *FinancialServices1* contains ten versions, between 557 (version 1) and 774 (version 10) features, and between 1,001 (version 1) and 1,148 (version 9) cross-tree constraints. We evaluated each evolution step on its own as well as the entire evolution timeline using the evolution-aware analysis of the TFM. To analyze the merged evolution timeline, we imported all versions and integrated them into one TFM. With these feature models, we can verify whether our implementation finds the same anomalies as one of the standard feature-modeling tools, FeatureIDE.

For the medium-sized real-world feature model, we use the feature model of the *Body Comfort System* product line [LLL+13] which we already used in Chapter 4. This feature model has been extended by multiple evolution steps by Nahrendorf et al. [NLS18]. The original version contains 28 features and in the last evolution step, it contains 49 features. To answer the research questions, we need to know exactly which anomalies exist and what their causing operations are. As anomalies only exist in presence of cross-tree constraints and as the original feature model of the case study does not contain any cross-tree constraints, it does not contain any anomalies [LSW15]. Thus, we manually create cross-tree constraints and evolution operations that seed anomalies. The fact that the feature model has already an evolution history is important so that we can verify whether we can correctly find all anomalies, their time of introduction, and evolution operations.

For the *Body Comfort System*, we seed 12 different anomalies at different points in time in the feature-model evolution timeline. In each of these scenarios, we deliberately introduce one anomaly. We systematically created four anomalies of each type, i.e., void feature model, dead feature, and false-optional feature anomalies. Anomalies may entail other additional anomalies and we consider these additional anomalies as well. For instance, if a feature of an alternative group becomes false-optional, all other features of that group become dead as a consequence. In particular, we create six dead feature anomalies and six false-optional feature anomalies. The manually seeded anomalies caused 11 additional anomalies. We document which evolution operations we performed in the DarwinSPL TFM editor for each evolution scenario.

Results To answer **RQ2.6**, we investigate whether all of our seeded anomalies including their additionally entailed anomalies are found and the correct time point of anomaly introduction is provided. For our case study, we can confirm that our tooling is able to identify all anomalies, to classify them correctly, and to provide the correct time point of anomaly introduction. All results are available in our online repository.

To answer **RQ2.7**, we need to verify whether the identified causing evolution operations in the explanation match those which we documented in the description for each evolution scenario for each anomaly. Moreover, the evolution operations of the explanations for the anomalies additionally caused by the seeded anomalies should be the same as for the seeded anomalies themselves. In the considered evolution scenarios, all identified causing evolution operations in the anomaly explanations matched the evolution operations performed in the editor. Other evolution operations identified as involved past operations also matched the operations we performed. Other irrelevant evolution operations for the anomalies are not listed. Our results indicate that we are able to identify all anomalies in the entire evolution timeline of a feature model and that we provide the correct evolution operations that lead to those anomalies. Thus, we can answer **RQ2.6** and **RQ2.7** positively.

Threats to Validity The internal validity is threatened as we manually analyze the *Body Comfort System* feature model to verify whether our method correctly identified all anomalies and causing operations. Thus, we might have misinterpreted which anomalies we introduced. To mitigate this threat, we use a feature model with a moderate size such that a manual analysis was feasible. We document all evolution operations we performed in the editor which we used as ground truth. Moreover, we verify that we identified the same anomalies as the most common feature mode analysis tool FeatureIDE.

The external validity is threatened as we only analyze three feature models of which one contains manually seeded evolution operations that create anomalies. We mitigate this threat by using two of the world largest available feature models with evolution, i.e., *Automotive02* and *FinancialServices1*. Moreover, the results of our implementation matched our manual analyses of the *Body Comfort System* feature model as well. However, it is not feasible to manually analyze the feature models of *Automotive02* and *FinancialServices1* to ensure that our method correctly detects all anomalies and causing evolution operations. We mitigate this threat by using real-world feature model evolution from different domains and sizes. Additionally, as already mentioned above, we cross-checked our results using FeatureIDE.

5.3.3. Performance and Scalability Evaluation

The applicability of our method to detect and explain anomalies in a TFM depends on its scalability for real-world feature models. Thus, to answer the second part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**, we need to investigate whether our method scales regarding performance. Besides, we are interested whether our method to analyze the entire evolution timeline at once and to reuse the respective solver provides a performance advantage compared to analyzing each evolution step individually. Thus, we pose two additional research questions that contribute in answering **RQ2**:

RQ2.8 How well does our method scale to large-scale real-world feature model evolution timelines?

RQ2.9 Does the analysis of the encoded feature-model evolution timeline (i.e., reuse of formula parts) improve the analysis performance?

Setup As subject systems to answer these research questions, we use the real-world feature models *Automotive02* and *FinancialServices1* (cf. Section 5.3.2). We deploy HyVarRec as a Docker container on virtual machines provided by an OpenStack private cloud. Each virtual machine uses Ubuntu 17.10, has four virtual cores and 8/16 GB RAM. We repeat each experiment five times and use the average values to reduce computation bias.

In the first iteration of our scalability evaluation, HyVarRec encodes all variables of the provided formulas as integer variables, i.e., also features as integer variables with a domain of $\{0, 1\}$. The reasoning behind this is that expressing certain constraints are easier, such as alternative groups. For instance, an alternative group with the parent feature P and the child features F_1, F_2, F_3 is encoded as follows using integer variables: $P = F_1 + F_2 + F_3$. Whereas the same elements are encoded as the following using Boolean variables: $(F_1 \leftrightarrow (\neg F_2 \wedge \neg F_3 \wedge P)) \wedge (F_2 \leftrightarrow (\neg F_1 \wedge \neg F_3 \wedge P)) \wedge (F_3 \leftrightarrow (\neg F_1 \wedge \neg F_2 \wedge P))$. Thus, we expected that we can exploit the capability of SMT solvers to handle integer variables to have more succinct formulas with fewer clauses.

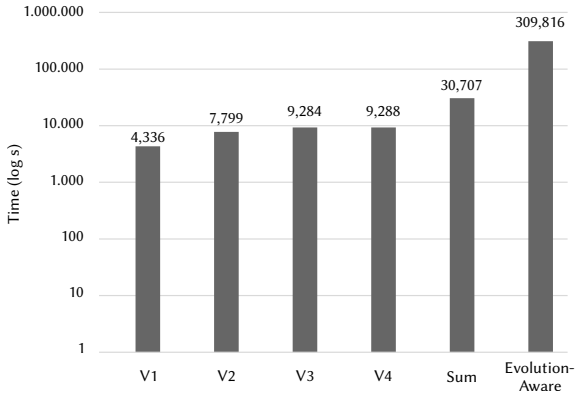


Figure 5.7.: Feature-anomaly analyses for four feature-model versions of Automotive02 using integer encoding (logarithmic scale).

However, as the results show, this was misjudgment and, consequently, we use the Boolean encoding in the second iteration of our scalability evaluation.

Results In the following, we only consider computation times of the HyVarRec backend and neglect the computation required by the `WebServer` to parse the input. This enables us to make statements about the performance and scalability of our method and reduces potential bias caused by our technology choices, i.e., that we deployed `HyVarRec` using a `WebServer`. This input parsing took for *Autmotive02* between $\sim$ 30 seconds (average for single evolution steps) and $\sim$ 79 seconds (average merged evolution timeline) and for *FinancialServices1* between $\sim$ 6 seconds (average for single evolution steps) and $\sim$ 23 seconds (average merged evolution timeline). All data and results can be found in our online repository (cf. Footnote 3).

As mentioned in the setup, in the first version of HyVarRec and our experiments, we use an integer encoding of the feature variables (i.e., instead of Boolean as variables, integers variables with the domain $\{0, 1\}$). Figure 5.7 shows the results of detecting all feature anomalies (i.e., dead and false-optional features) in the *Automotive02* feature model using an integer encoding on a logarithmic scale. The first columns (V1 – V4) show the computation times for analyzing the respective evolution step individually that is comparable to existing approaches, i.e., without creating a tagged formula and reusing the solver. The column labeled with "Sum" shows the summed values of all previous columns, i.e., the runtime that is necessary to analyze all evolution steps together but without making use of the tagged formula and resuing the solver. These results represent how existing approaches analyze feature models as they are not aware of evolution. The last column labeled with "Evolution-Aware" shows the results of our method, i.e., encoding the entire timeline in one query for the solver and reusing the solver for the analyses of the different evolution steps. The results show that the computation times were extremely high. For the single evolution steps, finding all feature anomalies took on average more than 2 hours. The summed up computation time to analyze all individual versions was more than 8 hours. For the merged model, it even took more than 86 hours.

Figure 5.8 shows the performance of detecting void feature model anomalies for *Automotive02* and using Boolean encoding of the feature variables. The average computation time for the void feature model analyses for each evolution step is $\sim$ 11 seconds. As can be seen, the sum of analyzing all individual evolution steps ($\sim$ 45 seconds) significantly exceeds the evolution-aware

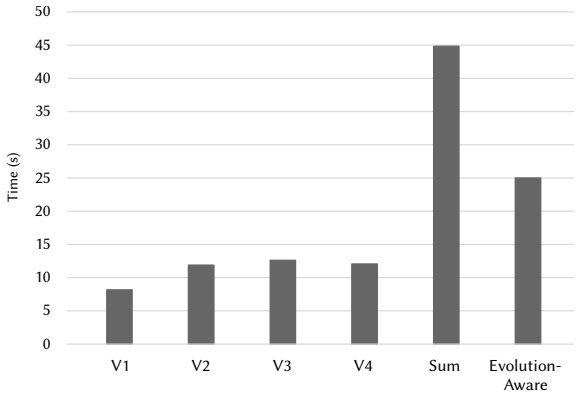


Figure 5.8.: Void feature model analyses for four feature-model versions of Automotive02 using Boolean encoding.

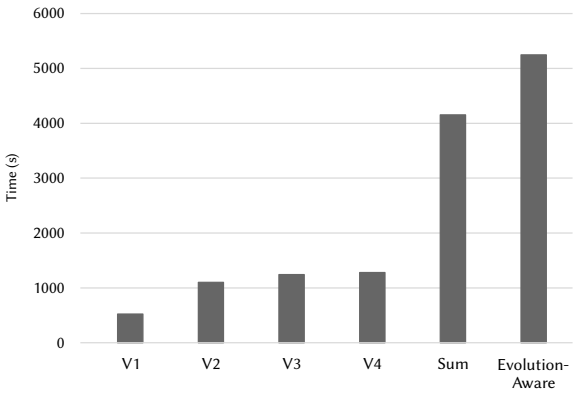


Figure 5.9.: Feature-anomaly analyses for four feature-model versions of Automotive02 using Boolean encoding.

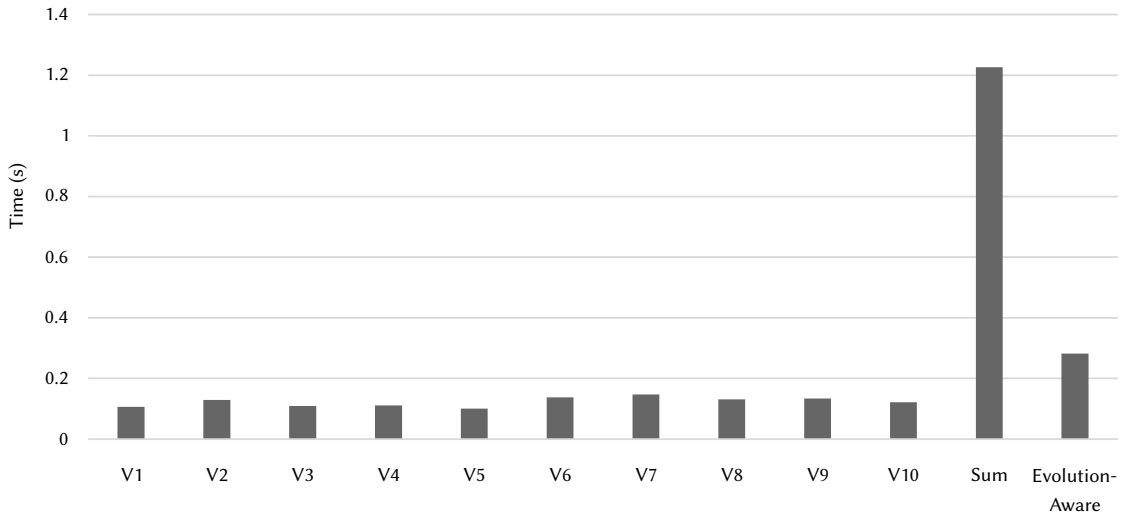


Figure 5.10.: Void model analyses performance for ten feature-model versions of FinancialServices01.

analysis ($\sim$ 25 seconds). Figure 5.9 shows the results for the feature-anomaly analyses for *Automotive02* using Boolean encoding. The average computation time for all evolution step is $\sim$ 17.30 minutes. The sum analyzing all individual evolution steps ($\sim$ 69.22 minutes) is less than the evolution-aware analysis ($\sim$ 87.42 minutes).

Figure 5.10 and Figure 5.11 show the results of the void feature model and feature-anomaly detection computation times for the *FinancialServices1* case study. The average computation time for each evolution step is $\sim$ 0.12 seconds (void feature model analysis) and $\sim$ 2.80 seconds (feature-anomaly analyses). Similar to the *Automotive02* case study, the sum of the individual computation times is significantly higher for the void feature model analyses compared to the evolution-aware analyses but for the feature-anomaly detection, the evolution-aware analysis is slower.

To answer **RQ2.8**, we can conclude that our method scales for large-scale real-world feature-model evolution. Even if we have computation times around 87.42 minutes (cf. Figure 5.9), it is an acceptable effort for analyzing the entire evolution timeline of such a large model. Typically, such an analysis is rarely performed and, thus, it could be run overnight. Compared to the most popular

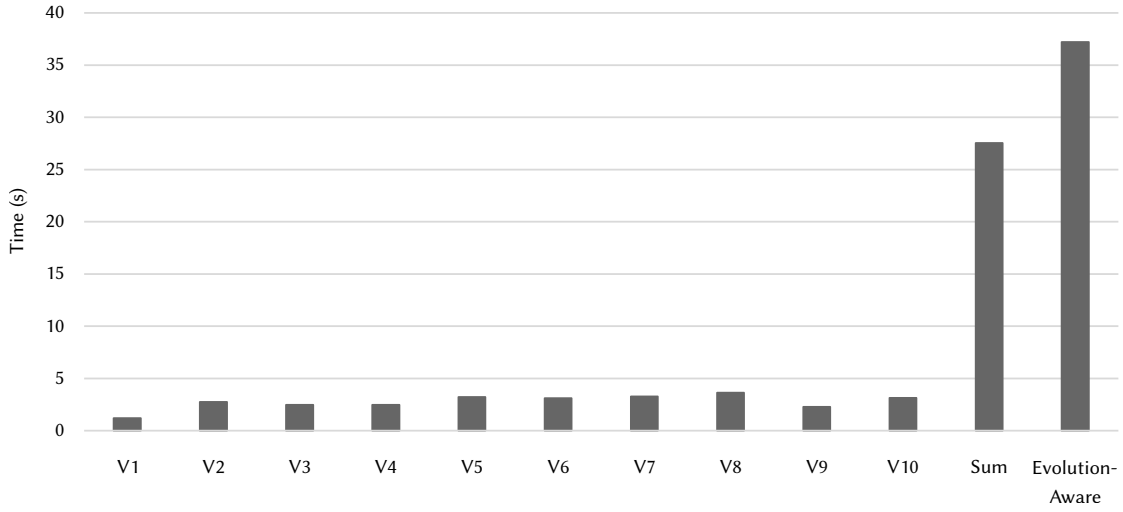


Figure 5.11.: Feature-anomaly analyses performance for ten feature-model versions of FinancialServices01.

feature-modeling tool suite FeatureIDE, this still takes more time. In FeatureIDE, checking each evolution step of the *Automotive02* for voidness takes a summed up computation time of $\sim$ 0.53 seconds and searching for feature anomalies in each evolution step takes a summed up computation time of $\sim$ 4.88 minutes. We believe that this is because FeatureIDE provides further optimizations and exploits the tree structure of the feature model. With the current version of HyVarRec, we are not able to do as we abstract from the tree structure after encoding it to a formula. Thus, HyVarRec is not aware of the feature tree, but only considers formulas.

For **RQ2.9**, we do not have an unambiguous answer. As the results show, in some cases, the evolution-aware analysis (i.e., for the merged model) is faster than performing the sum of all individual analyses. However, in particular, for the feature-anomaly analyses, our method is slower. As detecting anomalies is an NP-hard problems, we are not yet able to predict in which cases the evolution-aware analyses is faster. However, the factor for additional computation is not that high for the cases for which the evolution-aware analysis is slower, but for the cases for which it is faster, the difference is significant. In summary, the performance of our method scales for large-scale real-world feature models, but for some analyses our method is slower than existing methods. Nonetheless, our method provides additional information as we automatically retrieve of the anomaly introduction and determine how long each anomaly exists.

Threats to Validity The internal validity of this evaluation might be biased due to the encoding we use for the solver. We deliberately use a very naive encoding without any optimizations to have ground truth. Existing anomaly detection tools, such as FeatureIDE, implement further optimizations to speed up the analyses. In principle, we could use the same type of optimizations. We assume that even additional optimizations are possible that explicitly improve the encoding of feature-model evolution. Thus, we expect that our results are the worst case for our method.

The external validity might be biased as we only analyze the evolution timeline of two feature models. To mitigate this, we explicitly used real-world feature models with evolution that are the largest real-world feature models with the most evolution steps that we have access to. We assume that our method benefits the most if many evolution steps and many evolution operations are con-

tained in a feature model's timeline. Thus, we expect that if our method is applied in a long-term real-world feature model project, the results would even improve.

5.3.4. Evaluation of Explanation Complexity Reduction

As we illustrated in this chapter's introduction, anomaly explanations can grow very large for real-world feature models, even if single evolution operations were the cause. Thus, it can be very hard to keep a feature-model evolution timeline free from anomalies, as engineers need to understand an anomaly to fix it. We argue that if an anomaly has been introduced during evolution, evolution operations causing the anomaly are easier to understand and even more expressive than formula clauses. Additionally, the number of evolution operations is typically significantly fewer than the number of formula clauses of a standard explanation. In the second part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**, we investigate how to keep a feature-model evolution timeline free from anomalies. Thus, to investigate whether our method supports this task, we pose two additional research questions that contribute to **RQ2**:

RQ2.10 Can evolution-aware anomaly explanation reduce anomaly explanation complexity?

Setup As case studies, we use the feature models and their evolution timelines of the qualitative and performance evaluations (i.e., *Body Comfort System*, *Automotive02*, and *FinancialServices1*). In particular, we analyze the anomalies that have been introduced as part of evolution. For anomalies that existed from the first version, no evolution operations exist that introduced these anomalies and, thus, no explanation complexity reduction is possible. For the *Body Comfort System*, we analyze 17 anomalies as we also consider anomalies that arose due to other anomalies (cf. Section 5.3.2). Most anomalies of the *Automotive02* feature model have already existed from the first version and, thus, we cannot use them for this evaluation. As a consequence, only six anomalies have been introduced during evolution. In the feature-model evolution timeline of *FinancialServices1*, nine anomalies were introduced. As a baseline for the complexity reduction, we use the number of unsatisfiable formula clauses in explanations that other methods would provide. Between three to seven formula clauses are unsatisfiable for the anomalies of the *Body Comofrt System*, between four to 13 formula clauses are unsatisfiable for anomalies of *Automotive02* and between eleven to 98 formula clauses are unsatisfiable for anomalies of *FinancialServices1*. Despite the feature model of *FinancialServices1* being smaller than the one of *Automotive02*, the maximum number of unsatisfiable formula clauses is larger. This shows, that even smaller feature models can have large explanations.

To analyze to what percentage we are able to reduce explanation length, we compare the number of unsatisfiable formula clauses with the number of identified evolution operations causing the respective anomalies. Another approach one could imagine is for engineers to investigate all evolution operations that have been performed on the time point of anomaly introduction. To compare our method with reasoning about feature-model differences, we measure the percentage of identified evolution operations causing an anomaly compared to the number of all evolution operations performed at the introduction time point of the considered anomaly.

Results Figure 5.12 shows the relative explanation length of our method compared to "standard" explanations, i.e., formula clauses, and all evolution operations for the date of anomaly introduction. Lower numbers are better as this indicates shorter explanations compared to the other methods. For instance, a value of 20% would indicate that an explanation using our method would only be

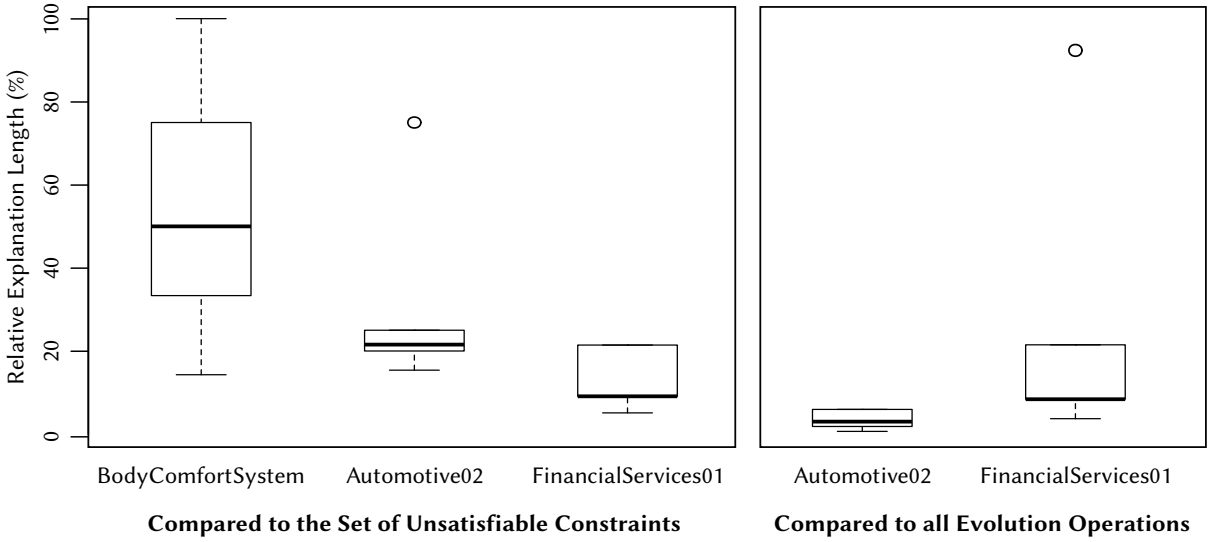


Figure 5.12.: Anomaly explanation complexity reduction rates.

20% as long as the explanation of another method. The first three plots compare the number of identified anomaly-causing evolution operations with the number of unsatisfiable formula clauses. The last two plots compare the number of identified anomaly-causing evolution operations with the number of all evolution operations performed that have been performed at the time point of anomaly introduction. For the latter comparison, we did not include the *Body Comfort System* as we explicitly performed single evolution operations that lead to anomalies (cf. Section 5.3.2).

The diagrams show that the relative explanation lengths compared with the number of unsatisfiable formula clauses for the *Body Comfort System* are between $\sim$ 14% – 100%, for *Automotive02* between $\sim$ 15% – $\sim$ 75%, and for *FinancialServices1* between $\sim$ 5% – $\sim$ 21%. The longest explanation contained 98 unsatisfiable formula clauses for *FinancialServices1* and we identified that only five evolution operations the respective anomaly. For two anomalies of the *Body Comfort System*, the number of identified evolution operations is equal to the number of formula clauses in the original explanation and, thus, no reduction is achieved. However, in nine cases of the *Body Comofort System* and in three cases of the *Automotive02* case study, we are able to reduce explanation complexity by more than half.

We achieve even more significant reduction rates for the comparison between causing evolution operations with all evolution operations. For *Automotive02*, the relative explanation length is between $\sim$ 0.6% – $\sim$ 6% and for *FinancialServices1* it is between $\sim$ 4% – $\sim$ 92%. The reason for the low relative explanation length for *Automotive02* is most likely that a very high number of evolution operations have been performed between the feature-model versions. The most significant reduction was achieved for the evolution step between version 1 and version 2 for which 169 evolution operations were performed and we identified 1 operation as the cause for a dead feature anomaly. In contrast, one anomaly in *FinancialServices1* was caused by almost all performed evolution operations (12 out of 13).

To answer **RQ2.10**, we show that the length of most of the explanations of the anomalies is significantly reduced by using our method. Moreover, we identified that typically only a small subset of evolution operations that have been performed is relevant for an anomaly explanation. Conse-

quently, we expect that it is significantly easier to fix anomalies using our method to explain the anomalies.

Threats to Validity The internal validity of this evaluation is threatened, as we compare the number of formula clauses of the original explanation with the number of identified evolution operations. Thus, we assume that understanding a formula clause of the original explanation is as complex as understanding an evolution operation. However, our experience with explanations has shown that understanding evolution operations is even easier for engineers than understanding formula clauses as the operations are common to engineers.

The external validity might be biased as we only analyze 32 anomalies in total. To mitigate this threat, we also use anomalies of the evolution of the largest two real-world feature-model evolution timelines we have access to. Moreover, when discussing the results, we distinguished between the anomalies from the *Body Comfort System* which we manually seeded, and the anomalies that stem from real-world feature-model evolution. As the results of the real-world case studies are similar to the one from the *Body Comfort System*, we expect our results to be representative.

5.4. Related Work

Feature model analysis is a widespread topic [SRC+12, BSR10, BTR05, TAK+14]. As satisfiability problems are the foundation for most feature-model analyses, we give a brief introduction to research dealing with this topic. In the subsequent paragraphs, we present different analyses for feature models that deal with or are similar to feature-model anomalies and their explanations. Additionally, we discuss analyses that explicitly incorporate information on feature-model evolution.

Satisfiability Problems in General Typically, feature models are analyzed using satisfiability problems (SAT) which are NP-complete [Joh92]. Thus, no deterministic algorithm exists that is able to find a solution in polynomial time [Wel82]. However, a solution of a SAT problem can be verified in polynomial time [Wel82]. In the literature, various heuristics have been devised that scale to large formulas in practice [GPF+96]. Satisfiability Modulo Theories (SMT) are a generalization of SAT [BT18]. In contrast to standard SAT problems, in SMT problems, formulas with integers and quantifiers can be solved as well. Guthmann et al. [GST16] and Liffiton et al. [LS08] provide methods to retrieve minimal explanations for unsatisfiable formulas. Guthmann et al. [GST16] compute the *minimal unsatisfiable core* using an SMT solver and Liffiton et al. [LS08] provide algorithms to compute minimal unsatisfiable subsets of formula parts. Using these techniques, no formula parts are contained in an explanation that is satisfiable and, thus, engineers directly see which formula parts are the reason for the unsatisfiability. However, these methods do not provide guarantees to retrieve a *minimum* explanation. As a consequence, such an explanation may not be reducible but another shorter explanation may exist. Similar to Guthmann et al. [GST16], we use the minimal unsatisfiable core of an SMT solver.

Feature Model Analyses Any feature model can be translated to a propositional formula [MWC09, BSR10] and, thus, SAT solvers can be used to analyze features models semantics, and most of the following approaches utilize SAT solvers. Multiple approaches are able to detect anomalies but do not provide any support in terms of explanations [Hem08a, ML04]. Other approaches are able to provide explanations for anomalies [KAT16, Bat05, LSW15, TBR+06, Tri12, FBG+13, KSR13]. Some of these approaches consider contradictions that occur during the configuration process [Bat05,

KSR13]. Lesta et al. [LSW15] provide a method to detect and explain dead features and false-optional features in attributed feature models. Trinidad et al. [TBR+06, Tri12] implement dead and false-optional feature detection and explanation in the tool suite FAMA which detects and explains dead and false-optional. The anomaly explanation method of Felferning et al. [FBG+13] does not relate explanations to the feature-model structure, as we do. Finally, Kowal et al. [KAT16] provide a method to detect and explain dead and false-optional features. They also detect redundant constraints and highlight which parts of an explanations are more important than others. Ananieva et al. [AKT+16] introduced a method to detect and explain implicit constraints in feature models. Using this method, it is possible to show only parts of a feature model to engineers which are (implicitly) related to a constraint or part of the feature-model structure. This can be used in combination with anomaly explanations to highlight related parts of a feature model. Kowal et al. [KAT16] and Ananieva et al. [AKT+16] also provide tool implementation in FeatureIDE. None of the previously mentioned methods incorporates evolution, neither for detection nor for explanation of anomalies. We could improve our explanation presentation by only highlighting affected feature-model parts in the feature diagram using the method of Ananieva et al. [AKT+16]. Moreover, we could integrate the method to detect redundant constraints.

Multiple techniques were published that are able to detect range inconsistencies in cardinality-based feature models [WLS+16, QPB+14]. A range inconsistency exists if it is not possible to find at least one configuration for each value of each cardinality. This is similar to dead and false-optional feature anomalies. For instance, if a cardinality of zero cannot be achieved in a valid configuration despite the domain allows this value, this is very similar to a false-optional feature. Similarly, if a cardinality that is bigger than zero cannot be achieved, this is similar to a dead feature anomaly. Moreover, severe range inconsistencies may lead to no valid configurations which are similar to a void feature model anomaly. In this chapter, we consider special cases of cardinality-based feature models (i.e., each feature has a maximum cardinality of 1). Quinton et al. [QPB+14] identified which evolution operations can lead to range inconsistencies. However, both approaches do not analyze feature-model evolution timelines and do not identify causing evolution operations.

Analyses Incorporating Feature-Model Evolution Only a few approaches exist that incorporate feature-model evolution in their analyses. As mentioned above, Quinton et al. [QPB+14] identified which evolution operations may lead to inconsistencies, but they do not analyze the operations or feature model versions.

Alves et al. present a theory for feature-model refactorings and a set of refactoring operations [AGM+06]. Similarly, Neves et al. propose a theory and a catalog for safe evolution templates [NBA+15, NTS+11]. However, the notions of refactorings of Alves et al. [AGM+06] and Neves et al. [NBA+15, NTS+11] only allow operations that do not remove valid configurations from the feature model but potentially add new ones. Thüm et al. [TBK09] present a more fine-grained categorization of feature-model changes. They distinguish between refactorings (the set of valid configurations remains the same), generalizations (the set of valid configurations is extended), specializations (the set of valid configurations is reduced), and arbitrary edits (valid configurations are removed but also new ones are added). Based on the previously mentioned categorizations, we could optimize our analyses such that only evolution operations are analyzed which can introduce anomalies.

Several techniques exist to reason about feature-model differences [DDP17b, BKL+16, TBK09]. These techniques can be used to derive changes between two evolution steps. To this end, Dintzner et al. [DDP17b] provide a set of operations they identified in the Linux kernel variability model which they are capable to detect with their tool FMDiff. However, this approach works only for KConfig variability models and none of these techniques is able to detect feature-model anomalies. Tartler et al. searched for anomalies in the variability model of the Linux kernel [RJW+09]. As the Linux kernel is one of the largest publicly available SPLs, the work has proven to be applicable to large-scale variability models. However, the method is specific to the Linux kernel variability model and does not incorporate evolution or provide anomaly explanations.

Schubanz et al. [SPP+13, SPB+12], Pleuss et al. [PBD+12] and Botterweck et al. [BPD+10] introduce EvoFM and the EvoPL framework. With their method, it is possible to model and plan feature-model evolution, similar to TFMs. They also provide techniques to check model and configuration consistency (i.e., only void feature-model anomaly). However, they do not incorporate evolution in their analyses, but analyze each feature model version individually.

Guo et al. [GWT+12, GW10] provide an approach to analyze feature-model consistency (i.e., void feature-model anomalies) in the presence of evolution. They do not analyze the entire feature models and their timeline again but only focus on parts changed by evolution operations since the last check. However, this requires that the feature model is valid before checking it again and, consequently, they only find the first anomaly. Moreover, they do not find feature anomalies and do not provide any explanations for inconsistencies. Nevertheless, it might be sensible to investigate whether their method can be combined with ours.

5.5. Chapter Summary

In this chapter, we addressed **Challenge 3: Detecting and Explaining Anomalies in Feature-Model Timelines** by answering **RQ2.6 – RQ2.10** which contribute in answering the second part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**. In particular, we provide a method that detects all anomalies in a feature-model evolution timeline by encoding the entire timeline in one satisfiability request for a solver. As fixing an anomaly requires the responsible engineers to understand the cause of that anomaly, we first identified at which time point an anomaly has been introduced and how long it exists and, second, we identify the evolution operations that caused an anomaly.

We implement our method in the tool suite DarwinSPL and qualitatively evaluate whether we are able to identify all feature-model anomalies and to correctly identify causing evolution operations. Moreover, we show that our method scales to real-world large-scale feature models and their evolution. We also investigate whether our encoding of the entire evolution timeline yields performance benefits. Our results show that in some cases we achieve better performance whereas in other cases, our analyses perform worse. Finally, we show that our method significantly reduces the complexity of anomaly explanations. Thus, we expect that fixing anomalies is easier for engineers using our method.

After modeling and planning feature-model evolution, we need to ensure its consistency and that it is free from design flaws. In Chapter 4, we provided a method to guarantee the structural consistency of a feature model which answers the first part of **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention**. The methods we present in this chapter are able to meet **Chal-**

lenge 3: Detecting and Explaining Anomalies in Feature-Model Timelines and together with the contributions of Chapter 4, we are now able to give the combinations of our methods as an answer for **Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention** in its entirety.

The contributions of this chapter raise several further research opportunities. To investigate the increase of comprehensibility and the support for fixing anomalies in the evolution history, we want to perform a supervised experiment with two user groups. As we highlighted in Section 5.3.2, the introduction of one anomaly might entail other anomalies. Thus, we want to integrate the detection of relations between anomalies (e.g., features that became dead because another feature became false-optional) or anomalies related to feature attributes (e.g., an attribute value that may never be selected). Another interesting question to answer is what defines the identity of an anomaly. To answer this question, we want to investigate the stability of anomaly explanations through the feature-model evolution. Moreover, we want to integrate existing optimizations that exploit the feature-tree properties for analyses and to visualize anomalies in the feature diagram. Finally, we want to investigate how to improve the feature-model evolution encoding and the exploitation of the knowledge about evolution. To this end, we also want to incorporate the usage of quantifiers in queries to solvers for the anomaly detection and measure its impact on performance.

With the contributions of Chapters 3 – 5, we are able to define consistent and anomaly-free feature-model evolution plans. After performing this feature-model evolution, configurations must be updated as well which yields additional challenges. In the next chapter, we will thus address **Challenge 5: Updating Configurations after SPL Evolution**.

Part IV.

Consistent Software Product Line Artifact Evolution

6 Augmenting Metamodels for Tracking and Planning of Model Evolution

The contents of this chapter are largely based on the work published in [NHS19, NHS+20].

Summary *SPLs are large-scale and long-living systems that undergo a long period of maintenance. Optimally, SPL evolution starts with the feature model. Other artifacts of an SPL, such as implementation models or feature-artifact mappings, must undergo a co-evolution with the feature model. In this chapter, we present a method for automatically augmenting modeling languages to enable capturing and planning of model evolution. Additionally, we generate an infrastructure that enables simplified access to model evolution data and enables seamless integration with existing tools. As a result, SPL engineers can use our method to augment modeling languages with uniform concepts to capture evolution which forms the basis for co-evolution of all SPL artifacts.*

Capturing past and planning future evolution of an SPL is necessary for structured, controlled, and well-considered development activities. As outlined in Chapter 3, feature models are a main communication artifact and lend themselves for planning SPL evolution. In Chapter 3, we devised TFMs – a modeling language to capture past and plan future feature-model evolution while supporting active development. However, a multitude of different feature-modeling languages exists that are already used in practice [BRN+13]. Thus, transferring the *temporal element* concept to integrate the expressiveness of TFMs into existing notations is desirable. Additionally, other SPL artifacts, such as documentation, implementation, or feature-artifact mappings must evolve in concert with feature models to meet new requirements. As SPLs in their entirety are a major strategic asset of companies, planning evolution of the respective artifacts is crucial in defining milestones for development, monitoring evolution progress and performing preliminary analyses for the planned state after evolution. Thus, methods to model co-evolution of all SPL artifacts in concert with feature models are required which is covered by **Challenge 4: Uniform Modeling of SPL Artifact Evolution**.

The current practice of using VCSs to capture model evolution does not support evolution planning and can emulate it only via workarounds, e.g., by maintaining an additional branch with a planned state of a model that has to be kept in sync by repeated merging, which may require manual resolution of merging conflicts. Figure 6.1 shows such a workaround for an example. In this example, t_0 is a past model state, t_1 is the current model state, and t_2 and t_6 are planned evolution steps. While t_2 is implemented, also an intermediate evolution step t_4 is implemented. However, with VCSs the concept of intermediate steps is not supported and, thus, a branch is created. At a

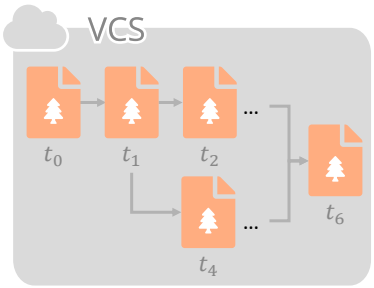


Figure 6.1.: Versioning and branching of model evolution using version control systems.

later point in time, this branch is merged with the original branch which may result in severe merging conflicts. The concept of *temporal elements* that we introduced with TFMs remedies these shortcomings. However, manually adopting the *temporal element* concept to existing feature-modeling or other artifact notations is very time consuming, prone to error, and repeated manual labor.

To address these shortcoming of current SPL evolution practices, we provide a method that generalizes the concept of TFMs and enables automatic adoption of *temporal elements* for other modeling notations. We focus on modeling notations as for each existing notation an equivalent modeling notation can be defined. We provide both the conceptual basis and a practical implementation to augment an existing metamodel with structures to store evolution similar to TFMs. This procedure is fully automated and, while it yields a new metamodel, we also generate facilities that ensure seamless integration with an existing modeling infrastructure, e.g., editors. Equivalently to TFMs, an augmented model consists of the entire evolution timeline in one sole artifact that preserves the temporal relation of the individual evolution steps. As a result, we provide a uniform language to capture evolution of SPL artifacts which forms the basis for consistent evolution. Thus, we address **Challenge 4: Uniform Modeling of SPL Artifact Evolution** and give first answers to **Research Question RQ3 – Consistent SPL Artifact Evolution**.

This chapter is structured as the following: in Section 6.1, we provide a set of transformation rules to augment a metamodel with evolution as first-class entity. In Section 6.2, we describe the automatic generation of access layers that support the (seamless) integration of models augmented with evolution and the existing infrastructure. In Section 6.3, we evaluate our augmentation method by providing an implementation and by applying our method to different real-world notations. In Section 6.4, we discuss related work that deals with model evolution. Finally, in Section 6.5, we give a summarizing overview of this chapter.

6.1. Augmenting Metamodels

Automatically augmenting metamodels with the concept of *temporal elements* requires transformation rules that can be applied using suitable tool support. In particular, each element of the metamodel whose instances may be subject to change is extended by a temporal element. This includes classes, attributes and also references between classes. We define a set of generic transformation rules to create an augmented metamodel. To ensure practical applicability, we provide transformation rules for all elements of an Eclipse Modeling Framework (EMF) Ecore metamodel, but our concepts are applicable to other Meta Object Facility (MOF) metamodel languages [Gro16]. We illustrate the transformation rules using an exemplary metamodel for state machines. Thus, we first

introduce the example metamodel. Afterward, we present the most relevant transformation rules in detail and elaborate on how they would work for the state machine. The remaining transformation rules work accordingly and can be found in Appendix A.

6.1.1. Exemplary State Machine Metamodel

State machines are models to capture system behavior on an abstract level in terms of states and transitions [HAR87]. Multiple state types exist: common states are the basic form to represent a system state; an *initial* state defines the start of a state machine's execution; if an *end* state is reached, the system execution stops; *composed* states are used to define hierarchical structures and consist of sub states, i.e., they are state machines themselves. A transition defines how a system state may change. To this end, each transition has a source state and a target state. A transition may be labeled with *triggers*, *guards*, and *actions*. Triggers of a transition define events a transition reacts to. Guards are expressions that evaluate to Boolean values and define under which conditions a transition may be activated. Actions of a transition are executed when that transition is activated and the system switches to the transition's target state.

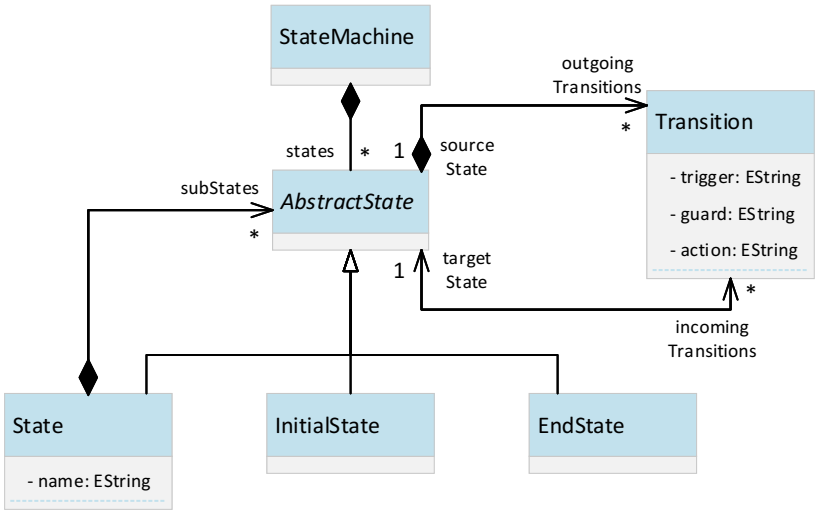


Figure 6.2.: Exemplary metamodel of state machines.

Figure 6.2 shows an exemplary state machine metamodel using the MOF (cf. Section 2.4). Each `StateMachine` contains a set of states, represented by the `states` reference to the abstract class `AbstractState`. For this reference, no lower or upper bound exists, i.e., an arbitrary number of states may exist in a state machine. Three state types exist that inherit from `AbstractState`: `State` represents common and composed states, `InitialState`, and `EndState`. To identify states, each `State` has a `name` attribute. If it is a composed state, its sub states are defined using the containment reference `subStates`. Outgoing transitions from an `AbstractState` are defined using the `Transition` class and the `outgoingTransitions` containment reference. An opposite reference `sourceState` of the class `Transition` can be used to determine the source state of a transition. As each state can have an arbitrary number of outgoing transitions, the lower and upper bounds are unlimited. However, each transition may only have exactly one source state, resulting in a lower and upper bound of 1 for `sourceState`. The references `incomingTransitions` and `targetState` are defined analogously. Triggers, guards, and actions of a transition are de-

fined as attributes using Strings. More sophisticated state machine metamodels would model this as individual classes, but we omit this for brevity.

6.1.2. Transformation Rules

In the following, we describe the transformation rules to augment metamodels with structures to store evolution and illustrate these rules using the state machine metamodel of Figure 6.2. We define these rules following the concept of *graph transformation rules* [AEH+99] which can be applied to metamodels as they are graphs. Each rule consists of a left-hand side and a right-hand side. The left-hand side defines structures that should be identified in a metamodel and that should be replaced. The right-hand side shows how the identified structures of the original metamodel should be replaced.

Augmenting Classes

During evolution, objects (i.e., instances of classes) are introduced or removed. For instance, a `State` of a state machine can be newly introduced at a certain point in time. Thus, it is necessary to be able to define the point in time of their creation or decommission. This can be directly modeled by using the temporal validity of a temporal element. Thus, an augmented class needs to become a temporal element. Figure 6.3 shows the transformation rule for augmented classes. After applying the transformation rule, `Class` inherits from `TemporalElement`. Consequently, each object of type `Class` has a temporal validity and can be created during evolution by setting its `validSince` or decommissioned by setting its `validUntil` respectively. Figure 6.4 shows this rule applied for the `AbstractState` class of the state machine metamodel. After application, `AbstractState` inherits from `TemporalElement` to support evolution of all types of states. If a new state is created at point in time t, its temporal validity is set to $\vartheta = [t, \infty)$. If a state is removed at t, the end of its temporal validity is set respectively, i.e., $\vartheta = [\vartheta_{since}, t)$.

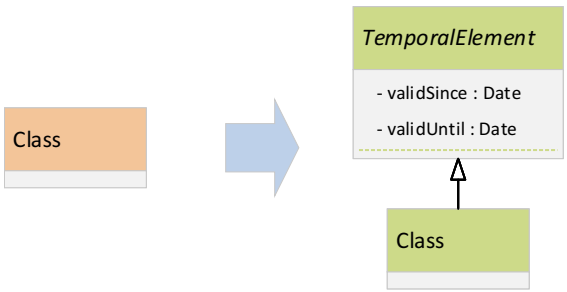


Figure 6.3.: Transformation Rule for Augmenting Metamodels with Class Evolution.

Augmenting References

Relations between objects are captured using references. As other model elements, values of relations may change as objects are added or removed from a relation. For instance, in the state machine metamodel, a reference `outgoingTransitions` between the classes `AbstractState` and `Transition` exists. Thus, `Transition` objects can be added or removed to that reference of an `AbstractState` object. However, if a `Transition` object is removed from that reference, knowledge that this transition was part of that reference at some point is lost. Thus, we need to persistently store reference values for each change during evolution.

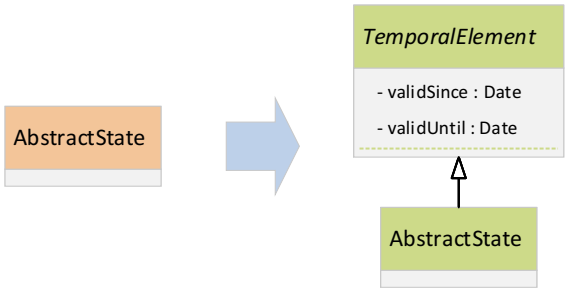


Figure 6.4.: Example of applying class transformation rule for the state machine metamodel.

In metamodels, references are no separate types and, consequently, we are not able to model them as temporal element using inheritance. To overcome this limitation, we create an individual association class. This association class wraps the original reference but is augmented to store evolution by inheriting from `TemporalElement`.

Figure 6.5 shows the respective transformation rule. In particular, we create a new class `TempReference`. This newly created class inherits from `TemporalElement`. The original reference of `Class1` is replaced by a reference to `TempReference` and a reference from `TempReference` to `Class2`. In the diagram, `Class2` originally has an opposite reference to `Class1` before augmentation. Thus, the transformation rule also creates opposite references from `Class2` to `TempReference` and from `TempReference` to `Class1`. Note that a `TempReference` class is created only once for all references of a type, i.e., `Class2` in the diagram. All references of that type that should be augmented then use this new class and have a reference to it.

In the original metamodel before augmentation, both references potentially have lower and upper bounds, i.e., l_1, l_2, u_1, and u_2. Over the course of time, objects that are referenced may change, i.e., referenced objects may be added or removed from that reference. Consequently, the total sum of objects that have ever been referenced by a single reference may exceed the upper bounds. However, metamodels are not able to express evolution information, i.e., lower and upper bounds for more than one point in time. Thus, the upper bounds are relaxed after augmentation and are replaced by unbounded references. The lower bounds are kept as is, since they must hold for a single point in time and, thus, it is not possible to reference fewer objects.

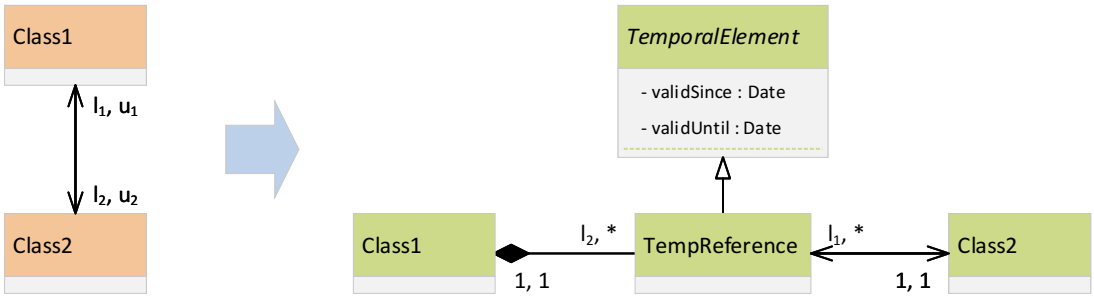


Figure 6.5.: Transformation rule for augmenting metamodels with unordered class-reference evolution.

Figure 6.6 shows the application of this rule for the `targetState` reference between `Transition` and `AbstractState`. To define an augmented metamodel that is able to store the evolution of a transition's target state, the new class `TempTargetState` is introduced that wraps the origi-

nal reference and inherits from `TemporalElement`. Additionally, a containment reference from `Transition` to `TempTargetState` is added and a reference from `TempTargetState` to `AbstractState` is added. To express the opposite `incomingTransitions` reference of the original metamodel before augmentation, references from `AbstractState` to `TempTargetState` and from `TempTargetState` to `Transition` are added.

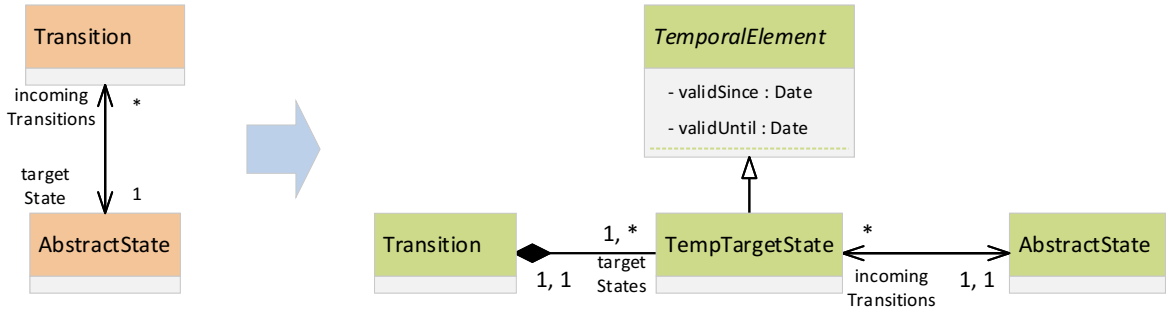


Figure 6.6.: Example of applying reference transformation rule for the state machine metamodel.

Reference evolution can be performed and planned by creating new objects of type `TempReference` and by setting their temporal validities. For instance, to change the target state of a transition at t, a new `TempTargetState` object is created, its temporal validity is set to $\vartheta = [t, \infty)$, and it is added to the transitions `targetStates` reference. Additionally, a reference from the new `TempTargetState` to the `AbstractState` that is the new target is set. If the transition had a target state before performing this evolution, the temporal validity of the respective `TempTargetState` is set to $\vartheta = [\vartheta_{since}, t)$. To retrieve a transition's target state for a given point in time t, first, all `TempTargetState` objects of the reference `targetState` are retrieved. Then, these objects are filtered by removing those objects that are not valid at t, i.e., they are not removed if $t \in \vartheta$.

Augmenting Attributes

Attributes of metamodels are similar to references. In contrast to references that capture relations between multiple objects, attributes are used to store values for primitive types, such as Integers or Strings. For instance, in the state machine metamodel, the `State` class has a String attribute `name`. Because of the similarity to references, metamodel limitations to capture evolution information for attributes are the same. If an attribute value changes, it is directly overridden. Thus, we need to introduce means to persistently store attribute values for each evolution step.

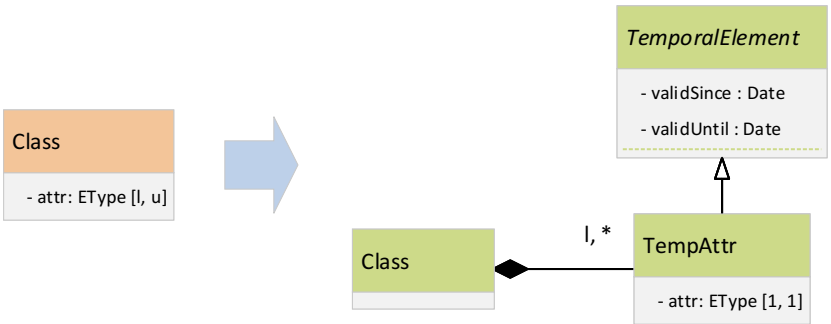


Figure 6.7.: Transformation rule for augmenting metamodels with attribute evolution.

Analogously to references, a wrapper class needs to be introduced to hold their evolution information. Figure 6.7 shows the transformation rule for capturing the evolution of values of the attribute `attr`. A new wrapper class `TempAttr` is created that holds exactly one attribute value of the same type as the original attribute `attr` before augmentation. Additionally, this class inherits from `TemporalElement`. In contrast to reference augmentation, `TempAttr` does not reference a second class. The value of a reference is an object of another class whereas the value of an attribute is a primitive value. Thus, this is captured by the attribute `attr` of `TempAttr` after augmentation.

Each `TempAttr` object represents an (evolved) attribute value. Consequently, the lower and upper bound of the new attribute `attr` of `TempAttr` is 1. The original lower bound before augmentation is captured by the lower bound of the containment reference of `Class` to `TempAttr`. Similar as for references, bounds for one sole evolution step cannot be represented in the static metamodel structure. Thus, the upper bound of that containment reference is unlimited.

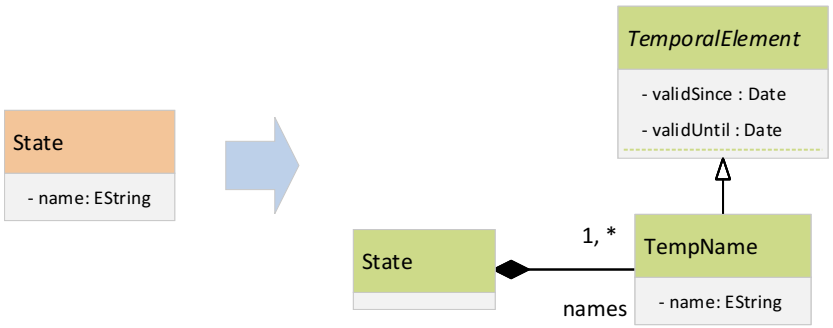


Figure 6.8.: Example of applying attribute transformation rule for the state machine metamodel.

Figure 6.8 shows the application of that transformation rule for the attribute `name` of the class `State`. A new class `TempName` is created that wraps a respective value as own attribute `name`. Additionally, this class inherits from `TemporaleElement`. The containment reference from `State` to `TempName` has a lower bound of 1, i.e., the original lower bound, and an unlimited upper bound. As already mentioned, this is due to the fact that a state name may change during evolution and, thus, multiple `TempNames` exist that are contained by the respective state but are temporally valid at different points in time.

Performing and planning attribute evolution works analogously as for references. For instance, to change a state name at point in time t, a new `TempName` object is created that has the new name as attribute value. The temporal validity of that `TempName` is $\vartheta = [t, \infty)$. The new `TempName` object is then added to the `names` containment reference of the respective state. If an old name existed, the temporal validity of that old name is set to $\vartheta = [\vartheta_{since}, t)$. To retrieve a state name for a point in time t, all `TempName` objects of the `names` reference are retrieved and all objects that are not valid at t are discarded, i.e., only the name is kept for which $t \in \vartheta_{name}$ is true.

Augmenting Ordered References

Typically, multi-valued references and attributes are unordered sets. However, it is possible to explicitly model ordered references and attributes. Consequently, this order can be subject to evolution which must be stored and maintained by augmented metamodels. To enable this, we extend the respective transformation rules to augment *ordered* references and attributes. For each evolu-

tion step, we store *one* particular list order. We realize this by using multiply linked lists. For each point in time, each temporally valid list node may only have one link to another node. Considering the entire evolution timeline, the order may change and, consequently, the link of a node must change as well. Thus, in the augmented metamodel, a node contains a set of links to successor nodes. Similarly, the ordered list itself has a set of root elements.

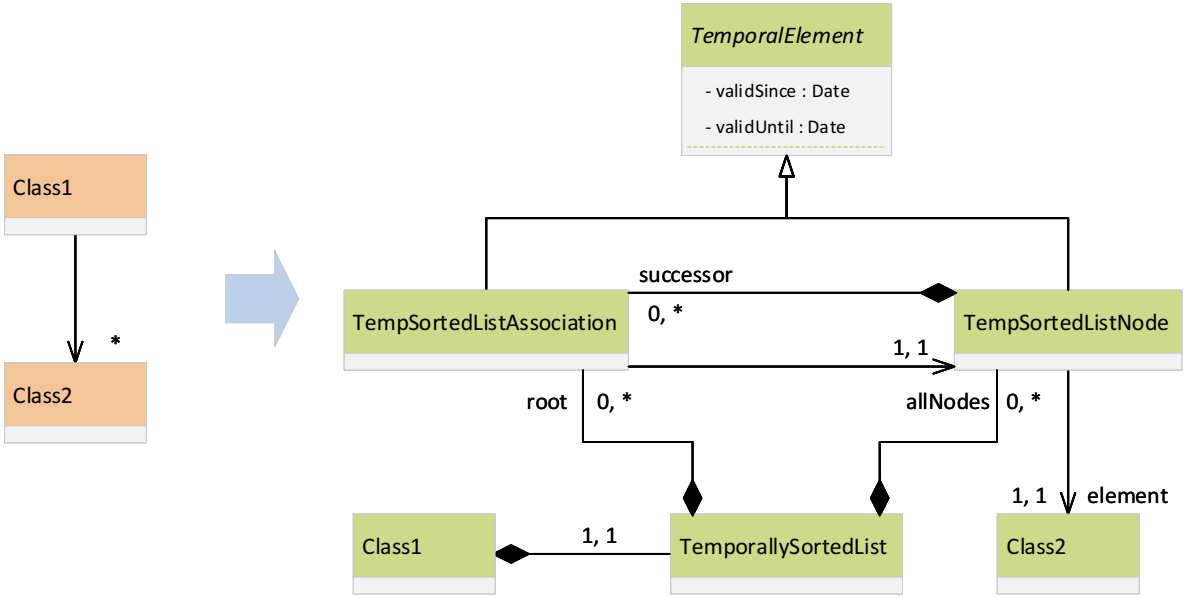


Figure 6.9.: Transformation rule for augmenting metamodels with ordered object-reference evolution.

Figure 6.9 shows the transformation rule for ordered reference evolution. For ordered multi-valued attributes, this works analogously and the respective transformation rule can be found in Appendix A. The values of a reference and their order are maintained by the new class `TemporallySortedList`. The values of the ordered reference are wrapped by the `TempSortedListNode` that represent nodes of the list. Each `TempSortedListNode` references exactly one object that represents the node value, i.e., the `element` reference. The class `TempSortedListAssociation` represents the links of the multiply linked list. For each evolution step, a list node has a maximum of one successor. However, as this successor may change during evolution, a `TempSortedListNode` references a set of successor links in the `successor` reference. Moreover, `TempSortedListAssociation` inherits from `TemporalElement` to represent change of orders by using temporal validities. `TempSortedListNode` also inherits from `TemporalElement` as entire list entries may be added or removed during evolution.

Non-Augmented Elements

Some elements of an original metamodel do not need augmentation. Attributes and references in EMF Ecore may be marked with additional properties, some of which preclude augmentation: References and attributes can be marked as *non-changeable* and, thus, they cannot evolve. Values of *volatile* and *transient* attributes and references are not stored at all. Finally, the value of *derived* attributes and references are calculated on demand and, thus, do not have an own identity. In consequence, these values are not subjected to evolution in a way that necessitates storing an evolution history.

6.2. Automatic Generation of Access Layers

In Section 6.1, we introduced transformation rules to augment a metamodel for storing evolution information. However, for a practical application of our method, we have identified three challenges that need to be addressed. First, as already pointed out, it is very tedious to manually apply the transformation rules for (feature) modeling languages. Second, while it is technically possible to alter and query evolution information directly within augmented models (e.g., for analyses or tools), accessing this information is cumbersome due to the structure of the augmented metamodel. Third, for existing modeling notations, tooling already exists which is not compatible with the augmented metamodel.

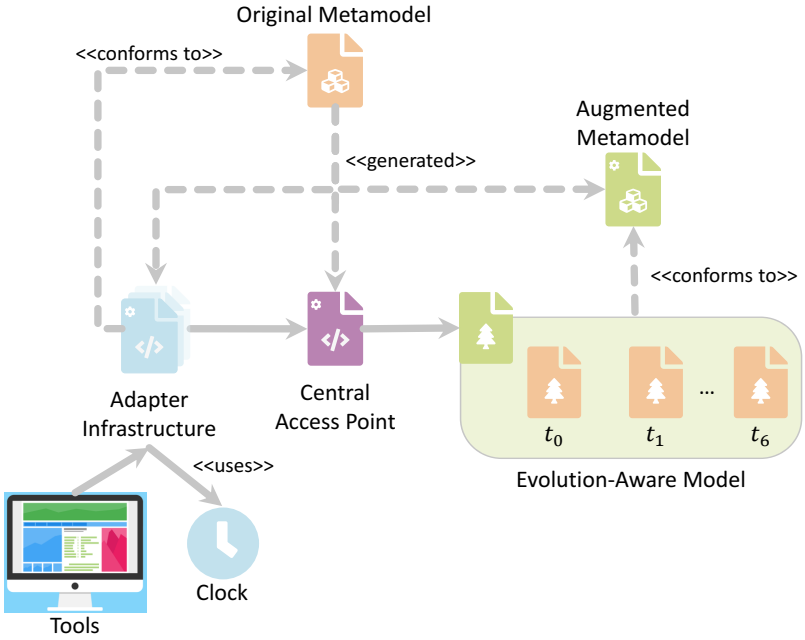


Figure 6.10.: Overview of the core contributions and their relations.

Figure 6.10 illustrates our three contributions which address these challenges: First, we fully automate the application of transformation rules to augment arbitrary metamodels. Second, in our generative method, we provide a central access point which hides the complex intricacies of augmented models to easily perform evolution and access respective information. Third, we generate an adapter infrastructure that is capable to maintain compatibility of the augmented metamodel with existing tools while automatically tracking performed changes as evolution by using the central access point and a time provided by a clock. With these contributions, we lower the barrier for both adoption and usage of augmented models.

6.2.1. Generating Augmented Metamodels

We devised the rules to augment metamodels with evolution in Section 6.1 in such a way that they are automatically applicable. Thus, to augment an existing metamodel, these transformation rules must be successively applied for each metamodel element.

For some model elements, information on (planned) evolution may be irrelevant, e.g., as they are not subject to evolution. Augmenting a metamodel element entails costs as creating and main-

taining additional augmented elements results in additional runtime and memory usage. To reduce this cost, we allow tailoring the augmentation process by only selecting those elements that should be augmented. For those elements, the respective transformation rule is applied. For the deselected and, thus, non-augmented elements, the respective elements of the original metamodel are copied without modification to the augmented model.

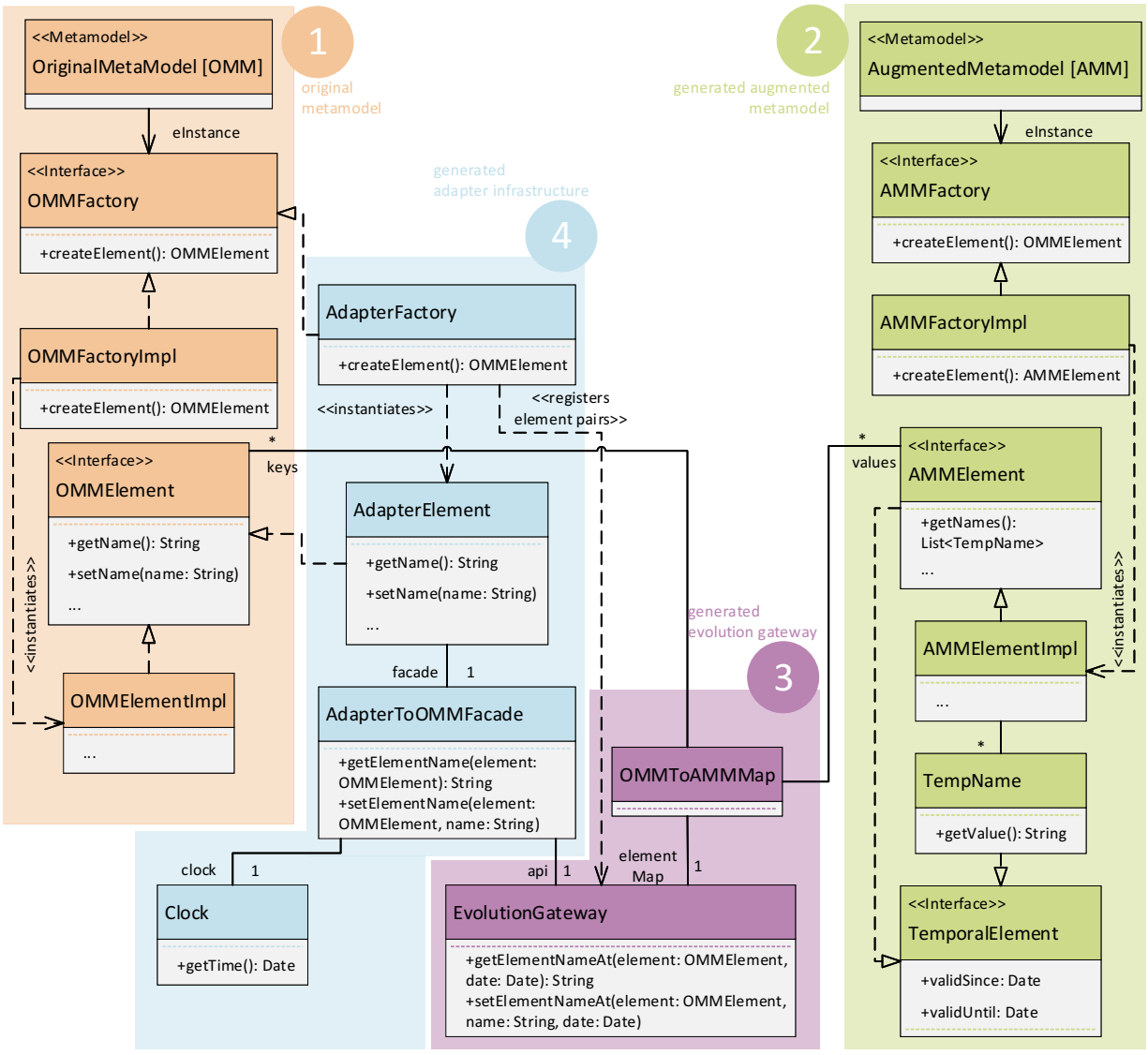


Figure 6.11.: Generated metamodel elements from original ① and respective augmented metamodel ②. Evolution gateway for centralized access to augmented models ③. Adapter infrastructure for seamless access using original metamodel interface ④.

In Figure 6.11, we illustrate how our automatic generation process works for an exemplary metamodel `OriginalMetaModel` (OMM) with one class `OMMElement` that has one attribute `name`. On the left side (① orange elements), Figure 6.11 shows the structures generated by the Eclipse Modeling Framework (EMF) for the original metamodel. The generated root class is the `OriginalMetaModel` that can be instantiated using an `OMMFactory`. Such a factory is responsible for cre-

ating all objects of a metamodel. In the basic structure, the interface of this factory is created together with a default implementation, i.e., the `OMMFactoryImpl` in this example. For each metamodel class, EMF creates an interface such as the `OMMElement` that provides methods to execute operations, and read and write attributes. For each of those interfaces, a default implementation is created, i.e., `OMMElementImpl` in this example. To create a new `OMMElement` object, an instance of `OMMFactory` is retrieved using the singleton pattern, i.e., the `eInstance` association in the diagram. Using this factory, the `createElement()` method is called that instantiates an `OMMElement`. For instance, the default factory `OMMFactoryImpl` creates an `OMMElementImpl`.

The right side (② green elements) shows the generated structure of the augmented metamodel after applying our transformation rules. The structures are similar to the one of the original metamodel. The main difference is that the `AMMElement` inherits from the class `TemporalElement` and, thus, has a temporal validity. Moreover, instead of having read and write methods for the `name`, the `AMMElement` returns a list of `TempNames`. Each instance of `AMMElementImpl` holds such a list. `TempName` inherits from `TemporalElement` as well and, thus, has a temporal validity. Moreover, each `TempName` holds one name that can be accessed using the `getValue()` method. In summary, we can store evolution for each `AMMElement` and for each `TempName`.

6.2.2. Seamless Usage of Evolution-Aware Models

Evolution of a model can be planned and performed by directly accessing the generated structures of the augmented metamodel. However, this is cumbersome as the generated structures are more complex than the structures of the original metamodel. For instance, to retrieve the name of an `AMMElement` that is valid at a specific point in time, the set of all `TempNames` must be retrieved using the `getNames()` method, users must iterate over those names, and extract the name that is temporally valid at the relevant point in time. This is complicated even for this simple example and becomes even worse if more complex structures such as augmented ordered references (cf. Section 6.1) are considered. Moreover, as our augmentation process yields a new metamodel, compatibility to existing tools or applications, e.g., editors or analyses, may be broken. This results in high adoption effort and, even worse, adoption may not be possible if third-party tools are used.

To remedy these problems and to increase acceptance of augmented models, we have devised a procedure that establishes compatibility between the augmented and the original metamodel by automatically generating a suitable *adapter infrastructure*. To ensure seamless usage of augmented models by existing tools, the interfaces of the original metamodel must be preserved. Figure 6.11 (④ blue elements) shows an exemplary adapter infrastructure we generate. When creating model elements of the original metamodel, a factory is used (cf. Section 6.2.1), e.g., an instance of the `OMMFactory`. By default, the `OMMFactoryImpl` is used for elements of the original metamodel. However, in the adapter infrastructure, we create an `AdapterFactory` that inherits from `OMMFactory` and overrides the default factory. Afterwards, the `AdapterFactory` is used by default.

Moreover, we generate an adapter element for each element of the original metamodel that inherits from the original metamodel element, e.g., the `AdapterElement` in the running example. These adapter elements serve as proxy for accessing the data stored in the augmented model elements. Thus, the methods for reading and writing data are overridden. These method calls need to be delegated so that the data of the augmented elements is accessed. As this procedure is similar to all elements of a metamodel, we generate a `AdapterToOMMFacade` using the facade pat-

tern [Gam95] that bundles this functionality for all adapters in one class. The facade provides mirrored methods for each method defined in the adapter classes. To access data of augmented elements, the facade needs to complement a time for which the access should be performed. Thus, the facade retrieves a point in time from a central `Clock`. We generate a default clock that uses the current system time. However, we provide an advanced override mechanism using the generation gap pattern [VV98], which enables to supply custom clocks, e.g., using a user interface. In Section 6.2.3, we discuss this clock mechanism and provide more details. The actual execution of an augmented model access is delegated to the `EvolutionGateway`, which is introduced in the next section.

In summary, we enable transparent usage of augmented models without changing existing implementations by using the overridden factory and the adapter elements. As a result, data that is retrieved is the temporally valid data for the time provided by the `Clock`. Changes performed to the model are automatically tracked as evolution. Additionally, providing support for planning evolution operations is simple as only the clock has to be overridden to enable selecting future points in time.

6.2.3. Model Access with the Clock Mechanism

The adapters serve as proxy for elements of the original metamodel to access elements of the augmented model. However, in the augmented model, the entire evolution is stored and, thus, multiple values that are valid at different points in time exist. Consequently, the adapter infrastructure needs to define a point in time for which data should be read or written. We define a `Clock` that provides a point in time for accessing augmented models. As it is not sensible to only access the model using the current real time, we enable to extend this `Clock` by providing an own time.

As default functionality, we generate a basic `Clock` that provides the current system time. This is particularly interesting if a stepwise migration to evolution modeling and analyses is envisioned. The main result is that changes automatically tracked as evolution history and analyses use the most recent model version. Thus, engineers specifying and analyzing the model do not see any difference. In addition, the model evolution history is not lost, but automatically saved in the augmented model. This data can be used in more advanced integration steps of the augmented model, e.g., if evolution analyses are performed.

More sophisticated use cases for augmented models encompass analyses of past model history and planning of future model evolution. Analyzing model history requires to set a point in time that lies in the past and planning model evolution requires to use a future point in time. Thus, this is not possible with the default clock implementation that always uses the current system time. To overcome this limitation, we provide an extension mechanism that enables to override the default clock and to implement custom clocks. For instance, a graphical user interface, such as for the Temporal Feature Model editor, can be implemented.

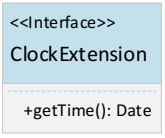


Figure 6.12.: Interface for Clock Extensions Setting a Time.

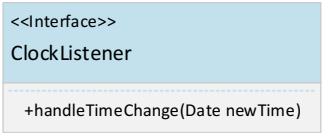


Figure 6.13.: Interface for Clock Listeners Being Notified About Time Changes.

The extension mechanism encompasses two extension types following the extension object pattern [Gam96]. Extensions of the first type enable *setting* the time and extensions of the second type enable *retrieving* the currently set time. Setting a time is possible by implementing the `ClockExtension`. Figure 6.12 shows the interface that respective extensions need to implement. Every time, a method on an adapter is called, e.g., to retrieve an attribute value or referenced elements, the `AdapterToOMMFacade` queries all registered extensions and uses the first time that is returned to access the augmented model. If no extension exists or, within a timeout, no extension returns a time, the default clock is used.

The currently provided time of the clock is important for all applications directly accessing the augmented model without the adapter infrastructure. Thus, we implement the observer pattern by providing a listener interface that is notified whenever the clock changes its time. Figure 6.13 shows the interface all listeners need to implement and which is used to register as listener extension. After registration, all listeners are notified whenever a clock changes its time. This can be used, e.g., to refresh the view of an editor.

6.2.4. Accessing Evolution-Aware Models

The `AdapterToOMMFacade` (cf. Figure 6.11, ④) is called by the adapters and retrieves the time of the `Clock` in order to execute the model access intended by the method call for a certain point in time. However, the `AdapterToOMMFacade` is unaware of the augmented metamodel and, thus, the method call is delegated to a layer that connects the adapters with the actual augmented model elements. We denote this connecting layer as `EvolutionGateway` (cf. Figure 6.11, ③). It is implemented using the facade pattern [Gam95]. The `EvolutionGateway` serves three main purposes: first, translating between adapters and augmented model elements; second, extracting temporally valid data for the time set by the clock; third, performing changes as evolution in the augmented model. In the following, we elaborate on how the `EvolutionGateway` addresses these three items.

Translating between Adapters and Augmented Model Elements The adapter infrastructure is only aware of the original metamodel and its respective interfaces. Thus, it does not know about the actual augmented metamodel and its elements. For seamless integration, each augmented model element needs a corresponding adapter. Moreover, methods are called on the adapters, but the actual execution needs to be performed on the augmented model elements. Thus, the `EvolutionGateway` needs to store the relation between adapters and their matching augmented model elements to correctly delegate method calls. This relation is established using a map (cf. `OMMToAMMMap` in Figure 6.11) with `AdapterElements` as keys and `AMMElements` as values. This map is held by the `EvolutionGateway`. When the `AdapterFactory` creates a new adapter, it registers this adapter using the `EvolutionGateway`. The `EvolutionGateway` for its part creates a corresponding `AMMElement` and stores both elements in the `OMMToAMMMap`. As a result, whenever a method call is delegated from an `AdapterElement` via the `AdapterToOMMFacade` to the `EvolutionGateway`, the `EvolutionGateway` queries the `OMMToAMMMap` to retrieve the `AMMElement` that is the mapped to the called `AdapterElement`.

Extracting Data from Augmented Model Elements Data of `OMMElements` is retrieved using provided getter methods, e.g., `getName()`. For each of those getter methods, the `EvolutionGateway` provides corresponding getter methods to extract that data from the augmented model. In particular, the `AdapterToOMMFacade` delegates the original method calls and enriches them

with the time provided by the clock. The actual execution of these calls in the EvolutionGateway is performed in three steps. First, the AMMElement mapped to the called AdapterElement is retrieved from the map. Second, the data that contains the entire evolution information from the augmented model is retrieved. For instance, if the original call was getName(), the EvolutionGateway would call the getNames() method of the AMMElement. As a result, it retrieves all TempNames that have ever been temporally valid for that AMMElement. Third, the value is extracted that is valid at the point in time provided by the clock. For instance, the retrieved set of names is iterated over and the TempName that is valid at the specified point in time is returned.

Performing Changes as Evolution Changes to data of adapters should automatically be captured as part of evolution. To perform changes, OMMElements and, thus, the respective AdapterElements provide setter methods. Similar as for reading data, the AdapterElements delegate these method calls to the EvolutionGateway which performs the actual data change as part of evolution. This is a four-step process. First, the AMMElement that is mapped to the called AdapterElement is retrieved from the map. Second, the value that is valid at the time t provided by the clock is retrieved and its temporal validity is set to end at t. Third, a new value element is created and its temporal validity is set to begin at t. Fourth, the newly created value element is added to the set of values of the AMMElement. For instance, if the name of an AdapterElement of Figure 6.11 is changed, the method setElementNameAt is called with the AdapterElement and a point in time t as parameters. Then, the EvolutionGateway retrieves the currently valid name $name_{old}$ as described in the paragraph before. Then, the validUntil of $name_{old}$ is set to t. Finally, a new TempName $name_{new}$ is created, its validSince attribute is set to t and $name_{new}$ is added to the list of names returned by the method getNames() of AMMElement.

Removing model elements in Ecore is performed using a utility method provided by the Ecore framework which unsets all references to the removed element. As a result, the element is not deleted, but it cannot be reached by other model elements anymore, and it is not persisted upon saving the model. Due to this process, it is undecidable for the adapter infrastructure or for the EvolutionGateway whether an element is temporally removed from a reference, e.g., to move it to another reference, or whether it is removed from the entire model. To provide remedy, the EvolutionGateway provides a distinguished method to remove elements from a model which results in setting the end of the temporal validity of the respective AMMElement.

In Ecore, changes to a multi-valued attribute or a relation are performed by retrieving the collection of the currently set values using the respective getter method and then performing changes on that collection. However, when using that getter method of an AdapterElement, the EvolutionGateway returns a collection of only those elements that are temporally valid at the clock time. As this is not the same collection as the one that is held by the AMMElement, changes to the returned collection are not transferred to the AMMElement and, consequently, these changes are not tracked as evolution. Additionally, inconsistencies would arise as the returned collection is modified but the actual data basis in the augmented model is not modified. To provide a remedy, the EvolutionGateway provides distinguished methods for modifying multi-valued (ordered) attributes and references. For instance, for an augmented state machine metamodel, adding an outgoing transition at a specific point in time to a state is possible by using the following method addOutgoingTransitionsAt(State state, List<Transition> transitionsToAdd, Date date). To provide seamless integration, all modifications to col-

lections that are returned by the `AdapterElement` method calls need to be delegated to the `EvolutionGateway`. To this end, all `AdapterElements` wrap the collections returned by the `EvolutionGateway` with an observer pattern. Upon modification of the wrapped collections, the owning `AdapterElement` is notified and it delegates the modification to the `EvolutionGateway`. Thus, consistency between those collections is ensured.

6.2.5. Summarizing Overview

Our method to augment metamodels with evolution and to provide an infrastructure to enable seamless integration with existing tooling comprises many building blocks that interact with each other. To recapitulate these different building blocks and how they are related, we briefly summarize them and illustrate their workflow. The first step is to generate the entire infrastructure. The only necessary input is the original metamodel and as a result, the augmented metamodel, the matching `EvolutionGateway`, and a corresponding adapter infrastructure is generated. Afterward, existing tools work without any modifications. Changes are tracked as evolution in the augmented model. This is possible as the central factory to create original model elements is overridden to create adapter elements instead.

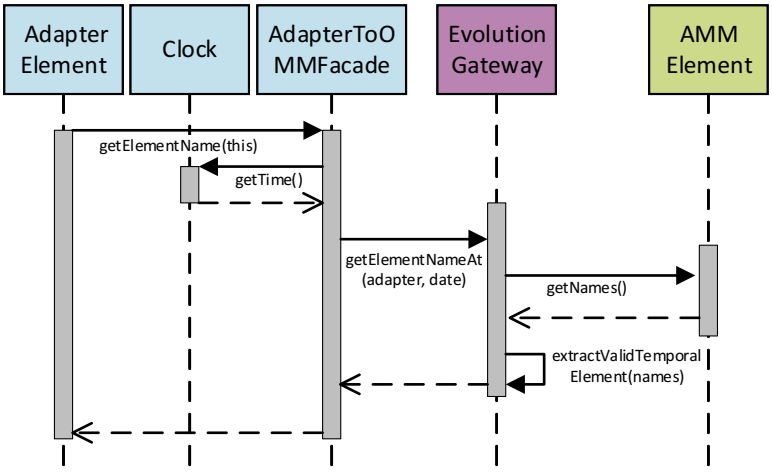


Figure 6.14.: Sequence of method calls for adapter object with usage of clock, facade, evolution gateway, and augmented model element.

Figure 6.14 illustrates a chain of method calls that enable seamless integration. As an example, we use the retrieval of a name of an `AdapterElement` using the method `getName()` (cf. Figure 6.11). The `AdapterElement` delegates this method call to the `AdapterToOMMFacade` using itself as a parameter. The `AdapterToOMMFacade` retrieves the clock time and delegates the method call to the `EvolutionGateway` with the `AdapterElement` and the time as parameters. The `EvolutionGateway` queries the actual data from the `AMMElement` by retrieving all names that have ever been temporally valid using the method `getNames()`. Afterward, the `EvolutionGateway` extracts the name that is valid at the point in time provided as a parameter. This name is then returned as result and passed to the `AdapterElement`, and subsequently to the caller of the `getName()` method.

6.3. Evaluation

Product lines are complex software systems that comprise variability models, implementation artifacts, and feature-artifact mappings. Thus, a multitude of different notations are used, e.g., feature models as variability models, UML class diagrams for modeling the static structure, or Java source for its implementation. Additionally, domain-specific models, such as Software Fault Trees (SFTs) for safety-critical systems [LG96], are used as well. To show applicability of our augmentation method for various kinds of models for product lines, we evaluate the different contributions of our approach for the all of the aforementioned notations. In particular, we show feasibility by presenting our implementation and illustrating how the augmentation process works based on a feature model notation. Additionally, we show that capturing and accessing model evolution timelines is possible. Finally, we show that our method enables seamless integration with existing tools and that it is applicable for large-scale real-world modeling notations.

6.3.1. Implementation

We implemented our augmentation method in a tool named `TemporalRegulator3000` that is open-source and accessible in an online repository[1]. The tool follows the generation process as elaborated in Sections 6.1 and 6.2. In particular, it generates an augmented metamodel by automatically applying the transformation rules, it generates the `EvolutionGateway`, and, finally, it generates the adapter infrastructure. To this end, it only needs an arbitrary EMF Ecore metamodel as input. For simplicity, the tool provides a wizard to guide engineers through the generation process. As augmented metamodels can significantly increase in size, the wizard enables to select which metamodel elements should be augmented and which should be left as-is. Thus, only relevant metamodel elements and elements that are subject to evolution need to be selected.

To illustrate `TemporalRegulator3000`, we utilize a metamodel for feature models. We created this metamodel based on the feature models that are used in one of the most common feature modeling toolsuites, i.e., FeatureIDE [2]. Figure 6.15 shows this metamodel. It consists of a `FeatureModel` with an `id`, a set of `features` and a `structure` defining the tree structure. `Structures` are defined hierarchically, i.e., each structure may have sub structures, at most one parent structure, and only one root structure exists. Additionally, each `Structure` references a feature for which it defines the position in the tree. Finally, each structure may have a `FeatureType` and a `GroupType`. Using a context menu, the `TemporalRegulator3000` wizard can be invoked. Figure 6.16 shows this wizard for the exemplary feature model. In this wizard, elements can be selected that should be augmented, and the augmentation process can be started.

During this process, a new Eclipse plug-in is created that contains the augmented metamodel. Figure 6.17 shows the augmented metamodel for the feature models. As each class, attribute, and reference is modeled as own `TemporalElement`, the augmented metamodel significantly grows in size. The detailed information is necessary to perform complex analyses, while the evolution gateway hides this complexity for common use cases.

[1] `https://gitlab.com/Adomat/temporalregulator3000`
[2] `https://github.com/FeatureIDE/FeatureIDE`

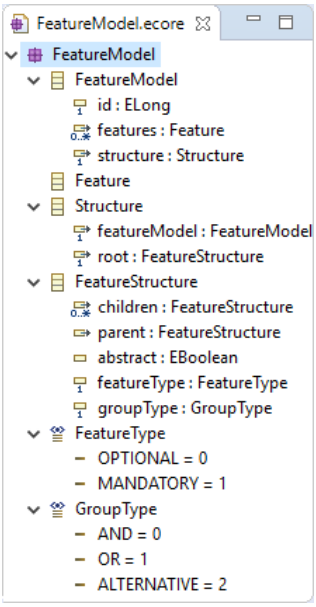


Figure 6.15.: An Ecore Metamodel for FeatureIDE Feature Models.

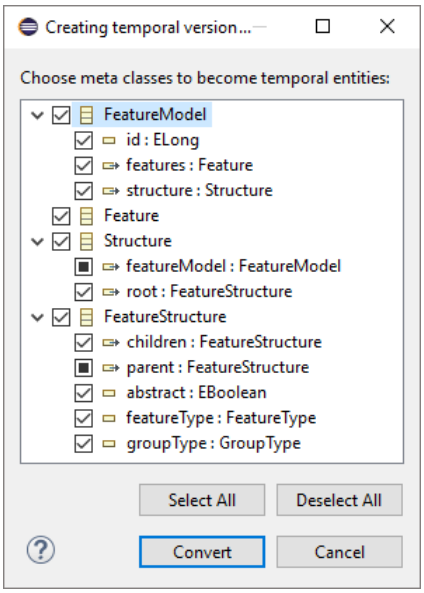


Figure 6.16.: Screenshot of the *TemporalRegulator3000* Wizard for a Feature-Model Metamodel.

6.3.2. Applicability to Real-World Metamodels

With our augmentation method, we enable to capture and plan an entire model evolution timeline for arbitrary modeling notations. Thus, we enable to model the evolution of feature models with all other SPL artifacts using the same language for evolution which addresses **Challenge 5: Updating Configurations after SPL Evolution**. With the evaluation of this chapter, we want to contribute in answering **Research Question RQ3 – Consistent SPL Artifact Evolution**. To this end, we pose the following research questions:

RQ3.1 In how far is the augmented modeling notation able to track, plan, and analyze model evolution in one artifact?

RQ3.2 To which extent is transparent use of the evolution-aware model possible while preserving compatibility with existing tools?

RQ3.3 Is the metamodel augmentation applicable to large-scale metamodels and does it scale?

To answer these research questions, we use the `TemporalRegulator3000` to augment multiple real-world modeling notations. To show the flexibility of our method, we augment a different modeling notation for each research question. In the following, we describe the experiments and results.

RQ3.1 To investigate whether we are able to track, plan, and analyze model evolution in one artifact, we utilize the evolution of a real-world feature model that we already used in Chapter 3, namely the *FinancialServices01* feature model history of a financial company. Table 6.1 recapitulates the available data.

The goal was to import all versions into one augmented model, verify that all versions are correctly imported, and perform analyses based on the evolution information. As a result, we extended

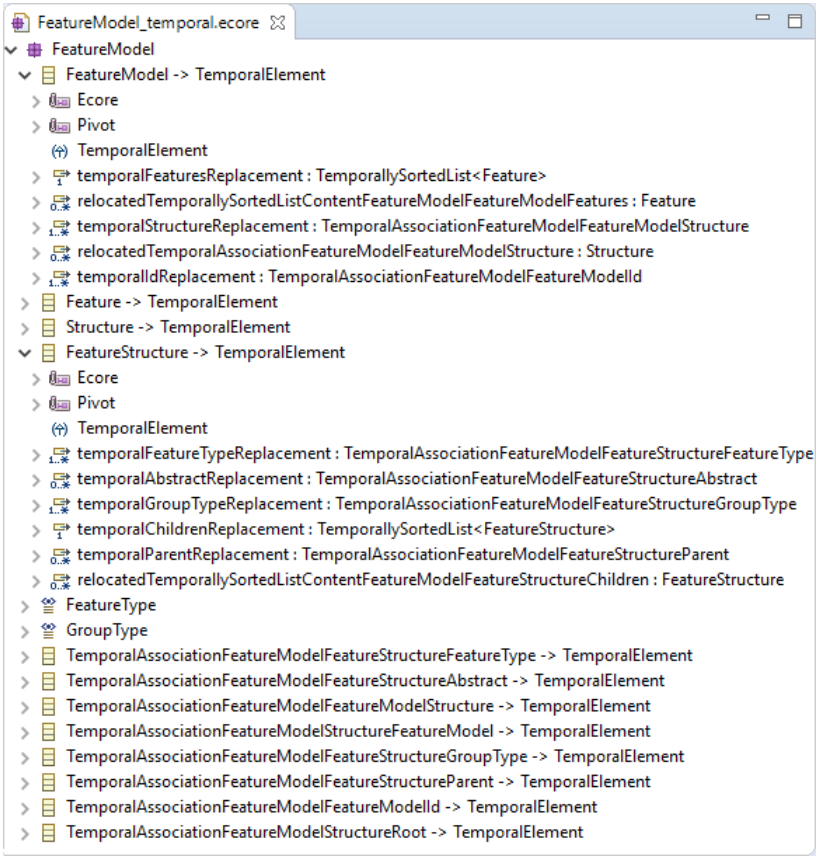


Figure 6.17.: The Augmented Metamodel for FeatureIDE Feature Models.

Table 6.1.: Numbers of Features and Groups of FinancialServices01 Feature Model Versions.

		V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
FinancialServices01	# Features	557	704	712	711	716	712	759	771	774	771
	# Groups	124	153	155	155	160	162	179	183	183	184

the `TemporalRegulator3000` by a generic importer that enables to import multiple model versions into one augmented model for arbitrary augmented notations. This importer only requires to implement a comparator for all elements of the original metamodel.

We augmented the used feature modeling language, implemented the previously mentioned comparator, and imported all ten versions of *FinancialServices01* into one augmented feature model. By manually setting the `Clock's` time to a future point in time, we imported multiple versions as planned evolution steps. To verify that the import worked as intended and to check whether accessing feature model versions for a given point in time is possible, we re-exported each feature model version and compared it to the original version. In summary, we were able to correctly track and plan the model evolution in one artifact.

To show that we are able to access model evolution data and perform analyses using that data, we investigated which feature groups changed frequently and which groups had many features added. The reasoning behind this kind of analyses is that these features are frequently affected from change

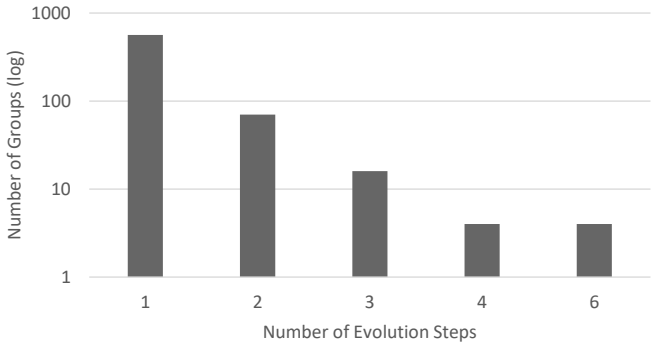


Figure 6.18.: Change Frequencies of Groups in the Evolution of the FinancialServices01 Dataset.

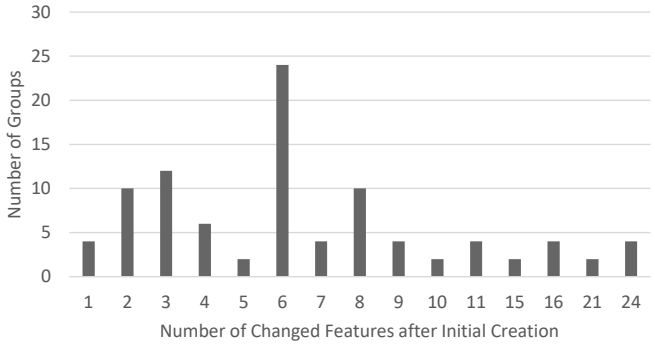


Figure 6.19.: Change Extent of Groups after their Initial Introduction in the Evolution of the FinancialServices01 Dataset.

and, thus, have a high potential for newly added defects. We retrieved this data by evaluating the parent-feature relationship entities between features and groups, and their temporal validities. Figure 6.18 shows the number of groups and the respective number of evolution steps in which they have been changed in logarithmic scale. Thus, most groups only change once, i.e., when they are introduced. Only four groups changed four times and four groups even changed six times. As feature groups are typically tested once after their initial introduction, we analyzed the extent of group changes after their initial introduction. The results of this analysis could be used to identify feature groups that should be re-tested. Figure 6.19 shows how many groups changed to which extent after their introduction. For most groups, six features have changed after their introduction, but for four groups, 24 features changed. Thus, they should be tested in more detail.

In summary, we were able to correctly capture an entire evolution timeline in one artifact. Moreover, we are able to access data regarding the evolution of a model without the need for computing differences between evolution steps. Thus, we can provide as answer for **RQ3.1** that tracking and planning of model evolution is possible for all metamodel elements. Additionally, we have shown that simple analyses are possible using this data and we are confident that more sophisticated analyses are possible as well.

RQ3.2 The augmentation model yields a new metamodel but, as already outlined, compatibility with existing tools is pivotal for the acceptance of our method. Thus, we generate the adapter infrastructure that preserves compatibility to the original metamodel while automatically keeping track of the evolution. To evaluate whether the generated adapter infrastructure enables

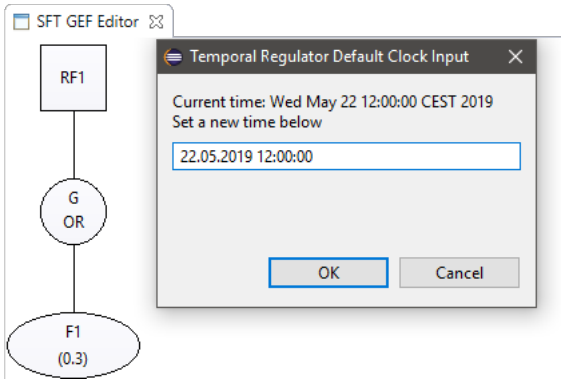


Figure 6.20.: Screenshot of the SFT editor before evolution.

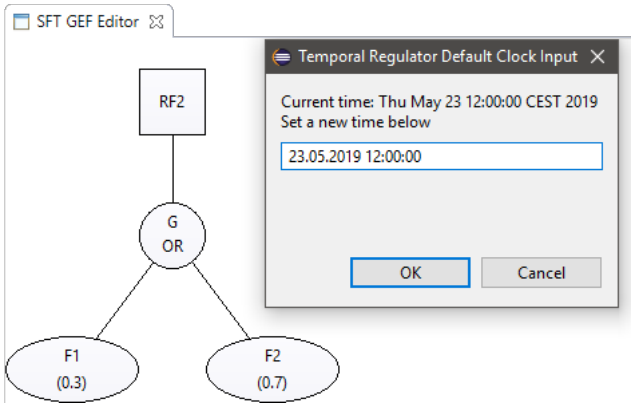


Figure 6.21.: Screenshot of the SFT editor after evolution.

seamless integration (**RQ3.2**), we augment an existing metamodel and create a model using an existing editor. In particular, we investigate whether the existing editor is still working without modification and whether evolution is automatically captured. We augment a Software Fault Trees (SFTs) metamodel and use its graphical editor with the augmented metamodel. To evaluate whether evolution has been correctly captured, we implement a simple user interface to set the `Clock's` time. However, this user interface is independent from the editor and just sets the time that is used by the adapter infrastructure.

Figure 6.20 shows the existing editor together with the user interface to set the `Clock's` time. In the first step, we set the time to May 22, 2019 and created the initial SFT with two faults (`RF1`, `F1`) and one gate (`G`). The editor worked without any modifications.

In the next step, we set the `Clock's` time to May 23, 2019 and create a new fault `F2`. Figure 6.21 shows these changes in the editor and the `Clock` user interface. The changes are capture by the editor in the metamodel. Subsequently, we are able to change the time back to May 22, 2019 and the editor correctly shows the state of Figure 6.20. When going to May 23, 2019, the editor shows the state of Figure 6.21. In summary, we are able to seamlessly use the augmented metamodel and to automatically store changes as evolution history. Thus, we can provide as answer for **RQ3.2** that seamless usage of editors that use the original facilities generated by Ecore to access models is possible.

RQ3.3 In **RQ3.1** and **RQ3.2**, we evaluate the functionality of our augmentation method. In **RQ3.3**, we evaluate whether this method works for large-scale real-world metamodels. To this end, we augment the official UML2 metamodel of Ecore and the JaMoPP metamodel representing Java [HJS+10].

Table 6.2.: Metamodel elements before and after augmentation.

		EClasses	EReferences	EAttributes
UML	Original	243	510	115
	Augmented	710	1167	109
JaMoPP	Original	237	105	15
	Augmented	303	270	14

We are able to successfully augment the UML2 and the JaMoPP metamodels. As augmented metamodels contain new structures to capture entire evolution timelines, the augmented metamodels significantly grow compared to the original metamodels. Table 6.2 shows the size of the augmented metamodels compared to the original metamodels for UML2 and JaMoPP. The number of classes grows significantly, especially for UML2. This is due to the fact that the augmentation method creates new association classes to capture the evolution of attributes and references. As a result, the number of references increases as well as each newly created association class needs a reference to the source and to the target of the original reference.

As Table 6.2 shows, the increase in number of classes is not the same as the sum of references and attributes of the original metamodels. Additionally, the number of attributes decreases. These numbers result from the fact that we use `TemporallySortedLists` with generic types to augment ordered references and attributes. Consequently, we add the classes required by the `TemporallySortedLists` only once to a metamodel and set the generic type dynamically. In summary, we are able to augment real-world large-scale metamodels that are used in industry. Thus, we enable to augment different metamodels used in SPL engineering and can answer **RQ3.3** positively.

6.3.3. Threats to Validity

Our evaluation is subject to internal and external threats to validity. The internal validity may be threatened as we evaluate whether we are able to track, plan, and analyze model evolution in one artifact by importing multiple feature-model version snapshots into one augmented model. This procedure works similar to differencing mechanisms and, thus, may result in inaccuracies. This threat can only be addressed if our method would be used in a long-term real-world study which is not feasible with limited resources. However, as we already outlined (cf. Section 6.1), we explicitly model structures that are able to store the evolution of all metamodel elements and, thus, we are more expressive than generic VCSs and differencing mechanisms.

A similar threat to internal validity is that we import feature-model versions as planned evolution steps. However, the versions of the original feature model were not planned as future versions. Thus, we could not verify whether real-world planning activity is supported by our method. Optimally, we would do this in a long-term real-world study as well. But we do not have the resources to do this with a partner that uses models as planning artifacts. As the mechanisms on the syntactical level are the same for planning and for capturing evolution, our augmentation method lays the technical basis for model evolution planning. Whether this planning is feasible in an industrial context is highly domain-specific and depends on other processes that are connected to the modeling.

An external threat to validity is that our generated adapter infrastructure does not enable seamless usage with all types of tools. This may be the case if custom methods to read, access, and write model files are used instead of the ones provided by Ecore. However, we integrate with the Ecore infrastructure and assume that this infrastructure is also used by the other tools. If custom solutions have been implemented, either changing our generation process to integrate with the custom infrastructure or changing the custom infrastructure to integrate with our generated structures is necessary. However, the general concept of our method should be applicable to all kinds of structures as the adapter infrastructure preserves the interfaces of the original metamodel.

Another threat to external validity is that we did not verify that our method is applicable to all existing metamodels. However, we applied our method a multitude of existing metamodels and also to metamodels used in industry. Moreover, we successively analyzed the Eclipse Modeling Framework (EMF) Ecore implementation and devised augmentation rules for each metamodel element type.

6.4. Related Work

Ample research has been conducted in the field of metamodel and model evolution. In this section, we present approaches that are related to ours. First, we describe approaches that enable to co-evolve metamodels and models. Second, we elaborate on research dealing with defining model evolution operations. Third, we discuss approaches that retroactively derive model changes between multiple versions or variants. Finally, we present research that is most similar to ours and addresses integrated capturing of model evolution.

Metamodel-Model Co-Evolution Model evolution has been subject to the research of many publications. A major focus has been put on metamodel-model co-evolution which considers the evolution of metamodels and how to keep their models consistent [Wac07, KSW16, GJC+09, HBJ09, RKP+10, CRE+08]. These methods define operations to modify models to upgrade them to the new metamodel version. However, the goal of these approaches is to restore compatibility of existing models with the new metamodel version. Thus, it is not possible to capture or to plan model evolution.

Model Evolution Operations Much research has been conducted in the field of defining operations to define model evolution timelines. Koegel et al. [KHH+09] and Nguyen et al. [NNP+10] introduce operation-based VCSs that store model modifications instead of model states in a VCS. Similarly, change-oriented programming synthesizes change objects derived from operations that have been applied to a model [EVC+07]. Engels et al. [EHK+02] and Kehrer et al. [KKT13] capture model evolution by means of pre-defined operations or transformation scripts.

Comparably, delta modeling enables to model differences between artifacts, such as models, in separate delta artifacts [CHS11, SBB+10]. In delta modules, delta operations are defined that enable to change a specific base model. This method is typically used to capture variability of artifacts but it can be employed to also capture model evolution. Seidl et al. devised the tool suite `DeltaEcore` that enables to define delta languages for arbitrary EMF metamodels [SSA14a]. With `DeltaEcore`, engineers can also define feature models and to map features to delta modules. Additionally, *evolution deltas* enable to express the evolution of model artifacts that can be associated to feature versions. As an extension to delta modeling, Lity et al. introduced *higher-order deltas* that enable to define changes to existing deltas in order to capture evolution [LKS16].

The aforementioned approaches rely on (delta) operations and, thus, the operations must be defined to enable model modifications. As we capture evolution by defining a temporal validity for each model element, we are independent of concrete operations. Additionally, we seamlessly integrate with existing tools and evolution is automatically captured. With the operation-based approaches, this is not possible without adaptation effort or only retroactively by deriving operations from multiple model versions.

Hermannsdörfer et al. devised a method to seamlessly capture model changes as evolution operations [HK10]. To this end, they defined a generic operation recorder that stores model changes in a separate operation model. This recorder uses the EMF observer infrastructure to be notified about model changes that are then captured as evolution operations in the operation model. While the capturing of model evolution works seamlessly, the retrieval of model versions does not. They do not provide an adapter infrastructure that enables access to different model versions without the need to change existing tools or analyses. Additionally, model versions can only be retrieved by applying all recorded evolution operations to the base model. Thus, for long evolution histories with many changes, this results in significant additional effort.

Retroactively Deriving Model Changes In the research fields of model differencing [BKL+12, BP08, Fou19, TEL+14] and reverse engineering [XS06, WRS+17], differences between multiple models (versions) are retroactively extracted. These methods are compatible with existing VCSs and no tooling needs to be adapted. However, computing the differences is expensive and often only an approximation. Additionally, model evolution planning is not considered by these methods.

Integrated Modeling of Evolution Gîrba et al. [GD06] introduce *Hismo* that introduces first-class entities to model evolution. In particular, they define `Histories` that contain `Versions`. To capture the evolution of a metamodel entity, such as a `State` of a state machine, two new metamodel elements are introduced: a `StateHistory` and a `StateVersion`. Each history contains multiple versions and each version knows its predecessor and successor version. To each version, a `Snapshot` is associated which is the actual metamodel element, e.g., a concrete `State`. Additionally, sub elements, such as a `Name` attribute of a state, also have `Histories` and `Versions`, e.g., `NameHistory` and `NameVersion`. A parent element history then contains the sub element history, e.g., the `StateHistory` contains the `NameHistory`.

Structurally, *Hismo* is similar to our augmented metamodels. Both store the evolution in an integrated way. The main difference is that we directly augment the original metamodel elements with temporal validities and that *Hismo* creates additional new classes. Thus, the expressiveness is comparable and similar evolution data can be retrieved. Some data is easier to retrieve using *Hismo*, such as histories of hierarchical elements, and some data is easier to retrieve using our augmented metamodels, such as concrete versions of elements. However, the *Hismo* approach is not devised for automatic tracking of evolution and seamless tool integration. In theory, this would be possible using a similar adapter structure as we propose but this is significantly more complex due to the newly introduced history and version elements.

6.5. Chapter Summary

In this chapter, we address **Challenge 4: Uniform Modeling of SPL Artifact Evolution** by providing a general approach to enable evolution modeling for arbitrary modeling languages. Consequently, we

are able to augment SPL artifacts consisting of feature models, realization artifacts, feature-artifact mapping. This forms the basis for consistent evolution of these artifacts and, thus, contributes in answering **Research Question RQ3 – Consistent SPL Artifact Evolution**.

With the tool `TemporalRegulator3000`, we provide an open-source implementation of our method that enables seamless integration in existing tool infrastructure. Thus, our method is non-invasive and can be applied without the need to change other tools or processes. As a consequence, evolution is automatically tracked, which provides the basis for analyses or planning tools that can be implemented at a later point in time. In our evaluation, we have shown the feasibility of our method and its applicability to large-scale real-world metamodels.

The contribution of this chapter raises several future research opportunities. First, we are interested in capturing co-evolution of multiple models using different modeling languages. By using the concept of temporal elements, all augmented modeling languages base on the same concepts to capture evolution. Thus, we are interested in generating entire evolution-aware development environments that provide further facilities to simplify modeling co-evolution, such as sophisticated viewers and editors. Additionally, we want to define general concepts to guarantee model consistency for augmented modeling languages in general, similar to TFMs in Chapter 4. To this end, an integration with consistency preserving edit scripts proposed by Kehrer et al. [KKT13] can be sensible.

In Chapter 3 – 6, we present how to model the evolution of SPL artifacts with a particular focus on feature models. SPL evolution has a direct impact on all configurations that have been used to derive products. In the next chapter, we propose a methodology to update configurations in accordance with SPL evolution which optimally preserves product behavior.

7 Guided Configuration Evolution

The contents of this chapter are largely based on the work published in [NST+20b, NST+20a].

Summary *Products of an SPL are generated by configurations consisting of selected features. Thus, changing feature models and their mapping to realization artifacts can lead to unintended changes to product behavior. We illustrate that updating configurations after SPL evolution requires knowledge of both, domain engineers responsible for SPL evolution as well as application engineers responsible for configurations. The challenge is that domain and application engineers might not be able to interact with each other. We provide a formal foundation and a methodology that enables domain engineers to guide application engineers through configuration evolution by sharing knowledge on SPL evolution and by defining automatically applicable configuration update operations. As an effect, we enable knowledge transfer from few domain engineers to an unlimited number of application engineers without the need to interact with each other. We evaluate four large-scale industrial product lines. The results of the qualitative evaluation indicate that our method is flexible enough for real-world product-line evolution. The quantitative evaluation indicates that we detect product behavior changes for up to* 55.3% *of the configurations which would not have been detected using existing methods.*

Real-world SPLs often have a high number of configuration options to fit users' requirements. For instance, customers can configure cars and their software through a web configurator of the car manufacturer [TKS18a] and the Linux kernel provides more than 21,000 configuration options [PTR+19]. A *feature-artifact mapping* uses Boolean formulas to associate features with reusable artifacts or parts thereof (e.g., through preprocessor statements in C++ code and a configuration sets variables for compile-time variability). Using these artifacts, a *product* can be generated automatically for a given configuration [ABK+16, CEC00]. In the SPL life cycle, two main roles are involved: during domain engineering, *domain engineers* specify feature models and feature-artifact mappings [PBL05]; during application engineering, *application engineers* define configurations to generate products.

Feature-model evolution typically serves as starting point for SPL evolution [PCA+13]. Other SPL artifacts such as realization artifacts, and the feature-artifact mapping are changed in concert with feature models [KGS18]. This can lead to unintended changes to the behavior of existing products [GTA+19]. For instance, Figure 7.1 shows an excerpt of the evolution of the Linux kernel running example. At t_0, the feature `Encfs` uses the encryption cipher AES as default. To provide more flexibility at t_1, the functionality for the AES cipher is extracted into an individual new feature and a new feature for the Twofish cipher is introduced. However, configurations selecting only `Encfs` represent different product behavior before and after evolution. Previous research identified that practitioners need to know how changes impact existing configurations and that knowing whether a system operates differently after evolution is crucial [BNR+14, MNM+18]. Thus, configu-

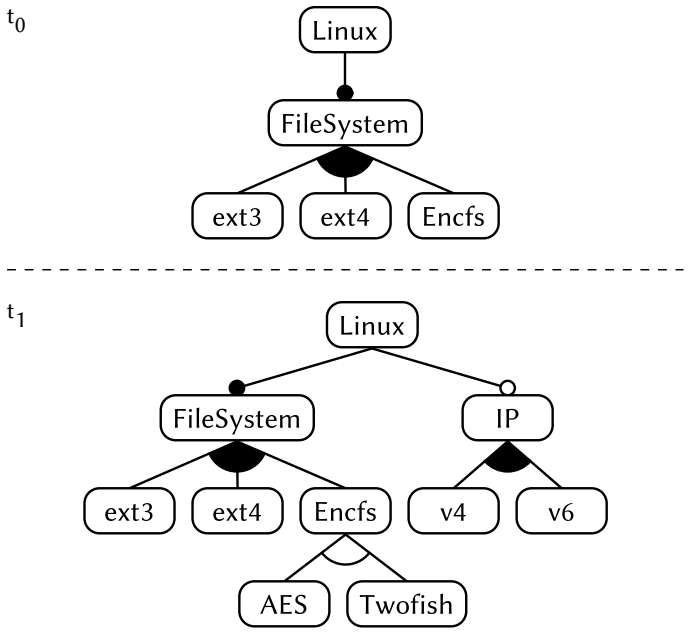


Figure 7.1.: Excerpt of the Linux kernel feature-model evolution running example.

ration evolution must be in line with SPL evolution which is part of **Challenge 5: Updating Configurations after SPL Evolution**. With current approaches, application engineers need to decide on their own how to update configurations used in the field without support from domain engineers which is time consuming and error prone [WSB+08].

Updating configurations to new SPL versions poses several challenges for domain engineers and application engineers to share their knowledge: first, detailed knowledge of the evolution may be lost as time spans between SPL and updating of configurations can exceed months or years; second, a communication barrier may exist as domain and application engineers may not know each other [Bos01]. For instance, a domain engineer modifying the Linux kernel does not know all end-users whose configurations are affected by the evolution. Hence, domain engineers are not necessarily aware which configurations are actually in use and may not know the requirements the generated products have to fulfill. Similarly, application engineers do not know the reasoning behind SPL evolution, whether their configurations are affected by that evolution, and the impact of the changes to their products' behavior. Thus, in isolation, application engineers cannot decide on how to change their configurations. Misconfigurations often arise due to application engineers being left with the task of updating their configurations [XZH+13, ZE14]. Previous research attempts provide automated fixes for configurations which may result in inadvertently altered product behavior [WSB+08, WPX+13, XHS+12, ZE14]. Moreover, these approaches assume that engineers in isolation are able to choose a suitable fix. However, depending on the evolution neither domain engineers nor application engineers are able to update configurations without knowledge of the other.

In addition to the communication barrier between the engineers, the number of application engineers typically is significantly higher than domain engineers. Consequently, even if domain engineers are able to communicate with application engineers, this leads to a high communication overhead or makes it even impossible [Bos01]. For instance, hundreds of domain engineers with different responsibilities are invovled in the development of the Linux kernel and thousands of ap-

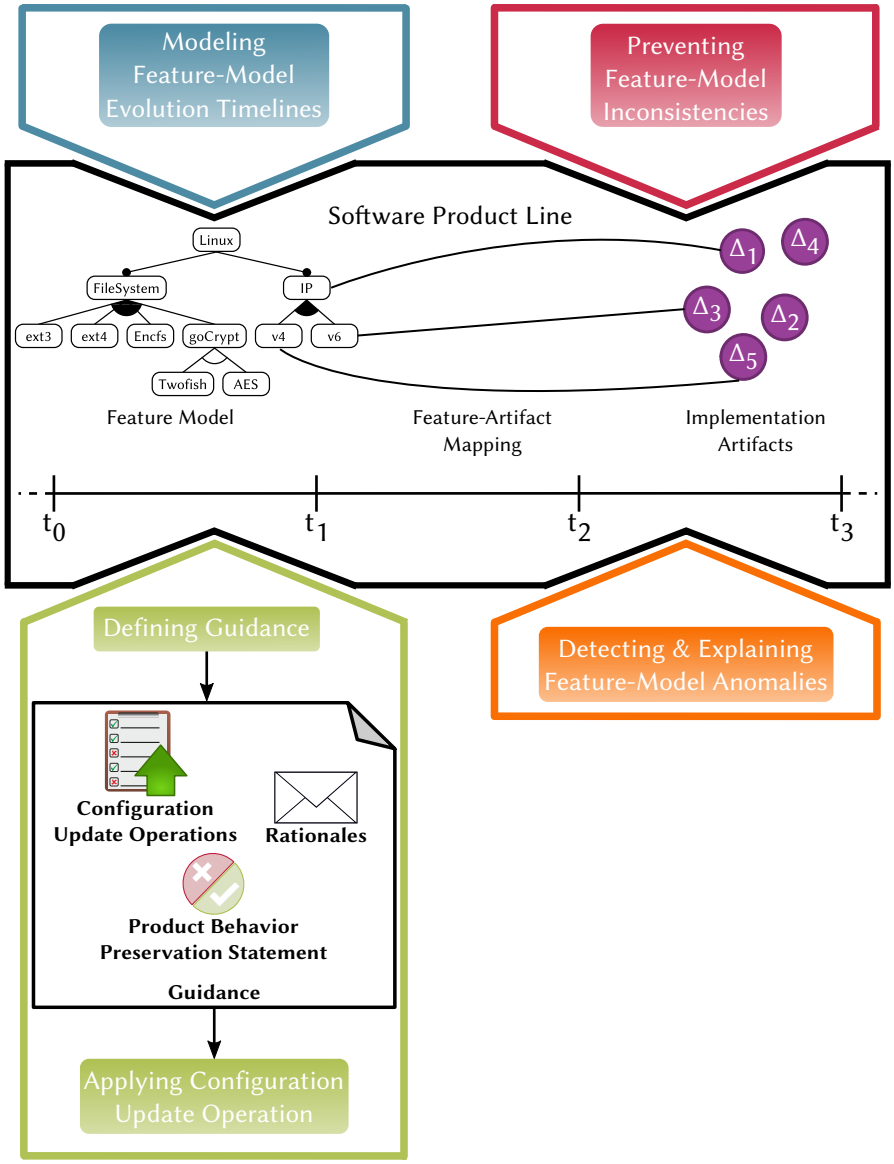


Figure 7.2.: Contribution overview – Step 4, Updating configurations after SPL evolution.

plication engineers configure it. To update configurations after evolution, each domain engineer would need to communicate with each application engineer. Moreover, industry reports that configuration logic changes almost weekly for some systems [BNR+14]. Thus, automating the communication between domain and application engineers is crucial as otherwise, updating configurations requires massive communication efforts which quickly becomes infeasible.

Figure 7.2 shows the contributions of this thesis with a focus on this chapter. After modeling feature-model evolution, ensuring its consistency, and improving its quality by fixing potential anomalies, other SPL artifacts need to evolve as well. In particular, we shows in Chapter 6 how the evolution of other artifacts, such as code or feature-artifact mappings, can be modeled. In this chapter, we present *guided configuration evolution*, a methodology for domain engineers to provide guidance for application engineers for updating configurations to a new SPL version. This contribution addresses **Challenge 5: Updating Configurations after SPL Evolution** by answering **Research**

Question RQ3 – Consistent SPL Artifact Evolution in terms of keeping configurations consistent with the evolved SPL. Our main goal is to enable knowledge transfer from domain engineers to application engineers without the need to talk to each other. To achieve this, domain engineers define guidance in terms of instructions for application engineers on how to update configurations to a new SPL version – ideally maintaining product behavior fully automatically. We propose a formal foundation and a general methodology allowing domain engineers to define and apply guidance. As basis, we assume an SPL approach with automated product generation using a configuration without any additional modifications applied to the resulting product. Domain engineers define guidance that consists of a rationale describing the SPL evolution and concrete operations for updating configurations that can be applied automatically. In the optimal case, product behavior is preserved after applying the configuration update operations. If product behavior perservation cannot be achieved, application engineers are made aware of that fact and can make an informed decision on how to adapt configurations. Guidance is defined once by domain engineers and can be used by an unlimited number of application engineers. In addition, domain engineers do not have to define guidance for each individual configuration, but can define it for large sets of configurations. To increase reuse, our methodology allows to define templates for guidance of typical evolution scenarios. We illustrate the use of our methodology by means of hree exemplary predefined evolution templates which address typical evolution scenarios. We perform three different evaluations: first, we formally prove that we are able to preserve product behavior of configurations for typical evolution scenarios using our methodology; second, we qualitatively evaluate whether it is feasible to apply and adapt our methodology to real-world SPL evolution; finally, in our quantitative evaluation, we determine the percentage of configurations for which we can a) automatically apply guidance, b) preserve product behavior, and c) detect changes to product behavior. The quantitative evaluation is split into two parts: i) We use real-world feature-model evolution and evaluate the domain engineer's perspective. ii) We use two real-world feature models with existing configurations and evaluate the application engineer's perspective.

In summary, we make the following contributions. In Section 7.1, we propose a formal foundation for domain engineers to express evolutionary changes to configurations. In Section 7.2, we define a methodology enabling domain engineers to guide application engineers in updating configurations. In Section 7.3, we provide three example evolution templates to support domain engineers, which illustrate the methodology and the formalism. In Section 7.4, we formally prove soundness of the templates by establishing behavior preservation for subsets of configurations. In Section 7.5, we describe how our methodology can be realized, which tools are necessary, and we introduce a prototypical tool *GuyDance*[1]. We qualitatively evaluate feasibility of guided configuration evolution in Section 7.6.1 and quantitatively evaluate guidance by analyzing evolution of real-world product lines and configurations in Section 7.6.2. In Section 7.7, we give an overview on related work. We conclude this chapter and give an outlook to future research directions in Section 7.8.

7.1. Behavior Preservation

During SPL evolution, features, artifacts, and feature-artifact mappings change. Consequently, a product that is generated for an existing configuration may behave differently than before evolution. To make statements about changes to behavior after evolution, we define our notion of product

[1] `https://gitlab.com/DarwinSPL/GuyDance`

behavior in the following. For simplicity of notation, we use the formalization of simple feature models (cf. Definition 3.1) instead of TFMs. We define that a configuration c is a set of selected features such that $c \subseteq \mathcal{F}$. Each feature that is not part of a configuration is implicitly deselected, i.e., $f \notin c, f \in \mathcal{F}$. As Figure 4.4 illustrates, we are able to project a feature model from a TFM using a given time point. Feature model projections and evolution operations can be used to synthesize a TFM again. Consequently, the presented formalisms and methods are also applicable for TFMs. To this end, also the feature-artifact mappings and configurations have to be extended by temporal elements. In particular, each entry of the feature-artifact mapping and each selection of a feature in a configuration have to be modeled as temporal element. Instead of removing, adding, or modifying a feature-artifact mapping, its temporal validity is set to the considered time point. If it is modified, a new feature-artifact mapping is then added with its temporal validity starting from the modification time point. Feature selections of a configuration have to be modeled as own entity inheriting from temporal element. If a feature becomes newly selected, a new selection object is created with its temporal validity starting at the time point of selection. This selection object is then added to the respective configuration. Deselection of a feature works by setting the end of the respective feature's selection object in a configuration to the considered point in time.

To generate a product of an SPL, all implementation artifacts that are mapped to the features of a configuration must be collected. The set $\mathcal{I}$ contains all implementation artifacts of an SPL. For instance, the `AES` feature can be realized using a plug-in `FileSystem.AES`. In a *feature-artifact mapping*, features are related to reusable artifacts [CECo0]. For instance, a feature-artifact mapping with preprocessor directives could look like: `#if AES <code> #endif`. We abstract from concrete implementation and feature-artifact mapping techniques. In Definition 7.1, we formalize our notion of mappings.

Definition 7.1: Feature-Artifact Mapping Syntax

Given a feature model FM and a set of implementation artifacts $\mathcal{I}$, a feature-artifact mapping $\mathcal{M}$ is defined as:

$\mathcal{M} : ac \rightarrow \mathcal{P}(\mathcal{I})$ assigning features in terms of an application condition ac to a set of artifacts $\mathcal{P}(\mathcal{I})$, with ac being a Boolean expression using features $f \in \mathcal{F}$ as literals.

Finally, we consider an SPL as a triple $SPL = (FM, \mathcal{I}, \mathcal{M})$. We denote all elements after evolution with a prime symbol, e.g., $\mathcal{F}$ evolved to $\mathcal{F}'$. Our notion of product behavior is defined by the resulting realization artifacts when evaluating a feature-artifact mapping using a configuration. Consequently, feature model constraints do not have an impact on product behavior and we do not consider them. Thus, for the sake of notation simplicity, we define feature-model evolution using standard set operations. For simplicity and without loss of generality, we assume that if a realization artifact $i \in \mathcal{I}$ is modified, this results in a new artifact $i' \in \mathcal{I}'$. As configurations are sets of selected features, we use common set operations to formalize configuration evolution and, thus, also configuration update operations. In Definition 7.2, we define our notion of feature-artifact mapping evolution.

Definition 7.2: Feature-Artifact Mapping Evolution

Given a feature-artifact mapping $\mathcal{M}$ and Boolean expressions exp, exp' with features $f \in \mathcal{F}$ as literals, feature-artifact mapping evolution is defined using the following replace operator:

$\mathcal{M}' = \mathcal{M}[exp \mapsto exp']$, with the replace operator iterating over all elements of $\mathcal{M}$ and replaces all occurrences of exp in the application conditions by exp'.

If a new artifact i' is added during evolution, it requires a new feature-artifcat mapping entry which is created using the following operator:

$M \oplus (exp, i' \in I')$ adds an entry to the feature-artifact mapping with the application condition exp related to the realization artifact i'.

To give an example, a feature-artifact mapping entry could look like: $\mathcal{M}(\texttt{v4} \wedge \texttt{v6}) = \{IP.multi\_protocol\}$. If the feature `Encfs` is deleted, we express this as: $\mathcal{F}' = \mathcal{F} \setminus \{\texttt{Encfs}\}$. The removal of the feature `v4` from a configuration c is expressed by $c' = c \setminus \{\texttt{v4}\}$. Finally,, if `v4` should be replaced by `v6` in a feature-artifact mapping $\mathcal{M}$, we express this as $\mathcal{M}' = \mathcal{M}[\texttt{v4} \mapsto \texttt{v6}]$.

In the following, we formalize product behavior and its preservation. As, in general, program behavior equality is undecidable [Ric53], we rely on a more conservative notion for comparison.

Definition 7.3: Product Behavior

For a product line $(FM, \mathcal{I}, \mathcal{M})$ and a configuration $c \in \mathcal{P}(\mathcal{F})$, the behavior of the resulting product is defined as:

$[\![\mathcal{M}]\!]_c = \bigcup_{ac}\{M(ac) \mid c \models ac\}$, denoting the set of artifacts resulting from evaluating the feature-artifact mapping $\mathcal{M}$ using the configuration c, i.e., all artifact sets assigned to feature-artifact mapping entries $m = (ac, \mathcal{I}_m) \in \mathcal{M}$ with c satisfying ac.

Product behavior of a configuration c is preserved if we can find a configuration c' that results in the same set of artifacts. Thus, we consider product behavior preservation as syntactic equality.

Definition 7.4: Product Behavior Preservation

For a product line $(FM, \mathcal{I}, \mathcal{M})$ evolved to $(FM', \mathcal{I}', \mathcal{M}')$, configurations $c \in \mathcal{P}(\mathcal{F})$, and $c' \in \mathcal{P}(\mathcal{F}')$, the product behavior of c in $\mathcal{M}$ is preserved by the product behavior of c' in $\mathcal{M}'$, iff

$$[\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_{c'}$$

For instance, if feature `v4` is mapped to $i_{\texttt{v4}}$, feature `v6` is mapped to $i_{\texttt{v6}}$, and feature `IP` is mapped to $i_{\texttt{IP}}$ and configuration $c = \{\texttt{v4}, \texttt{v6}, \texttt{IP}\}$ is used for product generation, the product behavior of c is defined by $[\![\mathcal{M}]\!]_c = \{i_{\texttt{v4}}, i_{\texttt{v6}}, i_{\texttt{IP}}\}$. During evolution, the features `v4` and `v6` are merged into `IP` and the artifacts $i_{\texttt{v4}}, i_{\texttt{v6}}$ are mapped to `IP`. By removing `v4` and `v6` from c resulting in $c' = \{\texttt{IP}\}$, c' preserves the product behavior of c (i.e., $[\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_{c'} = \{i_{\texttt{v4}}, i_{\texttt{v6}}, i_{\texttt{IP}}\}$).

Preserving product behavior after evolution may require a configuration to be updated using the respective evolution operators defined above. While product and configuration are often used synonymously in the literature, we adopt the distinction from the literature between those two

elements and consider a configuration as an implementation-agnostic set of features whereas a product comprises the implementation generated for a configuration [TKE+11]. As it is not efficient and does not scale to consider how to update each individual configuration, we reason on entire configuration subsets. To this end, we use the filter operator $\upharpoonright$ of Sampaio et al. [SBT16]. For a feature model $\mathcal{F}$ and a feature expression *exp*, $\mathcal{F} \upharpoonright exp$ yields the set of all configurations of $\mathcal{F}$ that satisfy *exp*. For instance, in the running example at t_0 if `Encfs` is deleted, all configurations that select this feature need to be updated, i.e., the configurations yielded by $\mathcal{F} \upharpoonright$ `Encfs`.

7.2. Defining Guidance for Updating Configurations

We provide a methodology that enables domain engineers in defining guidance for application engineers to update their configurations. Domain engineers express how and why the SPL evolved and define instructions for application engineers to update configurations in accordance with performed SPL evolution in a machine processable manner. These instructions can be applied fully automatically and ideally preserve a configuration's meaning in terms of product behavior – even if different features have to be selected. However, after certain SPL evolution operations, it is not possible to preserve product behavior. For such cases, domain engineers can define other possible configuration update operations and application engineers need to decide which of the suggested configuration update operations to apply to find a configuration that best suites their use case. Moreover, an SPL may yield new configuration options that were not accessible before evolution. For instance, in the running example, `AES` is extracted from `Encfs` at t_1 and a new cipher `Twofish` is added. Even if selecting `Encfs` and `AES` would preserve product behavior after evolution for configurations that originally selected `Encfs`, application engineers might want to use `Twofish` instead. Thus, even if product behavior can be preserved, application engineers might want to select different update operations. Applying guidance is time independent from SPL evolution and application engineers can use guidance to update configurations that are relevant to them.

Figure 7.3 shows the general idea of our contribution. For an SPL, application engineers derive a configuration c and a product represented by $[\![\mathcal{M}]\!]_c$. After domain engineers perform SPL evolution resultin in SPL', they define guidance which can be used by application engineers to update their configuration to c' and corresponding product $[\![\mathcal{M}']\!]_{c'}$, which can be derived from SPL'. Depending on the evolution operation and configuration updated operations, the defined guidance may preserve product behavior. However, domain engineers always have to state whether product behavior is preserved, whether it is not preserved, or whether it is unknown. Our notion of product behavior preservation is conservative as we assume that the resulting implementation artifacts have to be the same after evolution (cf. Section 7.1).

7.2.1. Structure of Configuration Evolution Guidance

Configuration evolution guidance consists of a brief description of SPL evolution, configuration update operations, and statements of product behavior preservation. Table 7.1 shows an example of guidance for a *Delete Feature* evolution operation. We added identifiers in brackets in the table which we refer to in the text. The rationale of the SPL evolution is defined in natural language (i.e., r in Table 7.1). This serves as basis for application engineers to understand the overall scope and reasons for the SPL evolution. Second, domain engineers define a set of guidance elements (i.e., $\mathcal{X}$). Each guidance element ($x_i \in \mathcal{X} = (S_i, \mathcal{U}_i, t_i)$, visualized as row in Table 7.1) is defined for a configuration

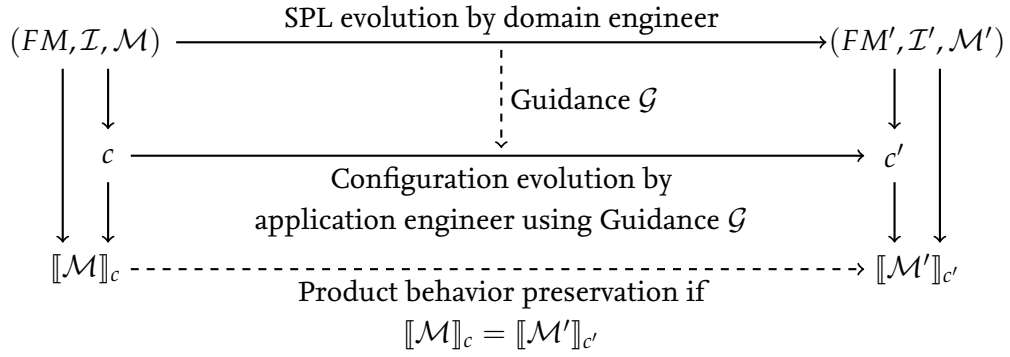


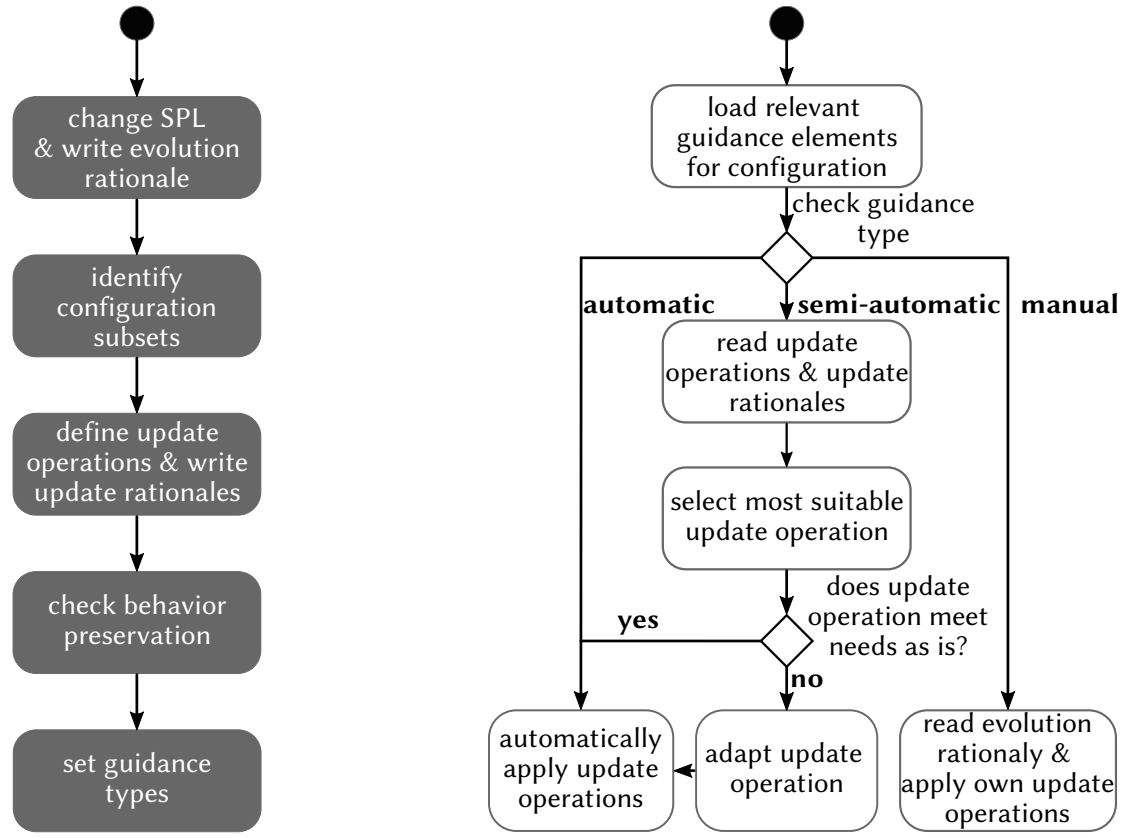
Figure 7.3.: Updating a configuration after SPL evolution.

Table 7.1.: Guidance for a *Delete Feature* operation.

	Operation: Delete feature f_0 with realization artifacts (r) $\mathcal{F}' = \mathcal{F} \setminus \{f_0\}, \mathcal{M}' = \mathcal{M}[f_0 \mapsto \mathit{false}]$				
	Configuration Subsets	**Update Operations**	**Preserves Behavior**	**Update Rationale**	**Type**
(x_1)	$\mathit{Delete}_0 : c \in \mathcal{F} \upharpoonright \neg f_0$ (S_1)	$c' = c$ $(op_{1,1})$	yes $(b_{1,1})$	Not affected. Can be left as-is. $(r_{1,1})$	autom. (t_1)
(x_2)	$\mathit{Delete}_1 : c \in \mathcal{F} \upharpoonright f_0$ (S_2)	$c' = c \setminus \{f_0\}$ $(op_{2,1})$	no $(b_{2,1})$	Remove f_0 from all configs. $(r_{2,1})$	semi-autom. (t_2)

subset (S_i). Thus, domain engineers can define one update operation for large subsets of configurations instead of defining an update operation for each individual configuration. The set of configuration update operations ($\mathcal{U}_i$) for each guidance element are suggestions for application engineers on how to update their configurations. For each configuration update operation ($u_{i,j} \in \mathcal{U}_i$), domain engineers need to specify the concrete set operation on the configuration ($op_{i,j}$), a rationale ($r_{i,j}$) in natural language defining why they defined this operation and in which cases it could make sense to be applied. Additionally, domain engineers specify whether product behavior is preserved ($b_{i,j}$) by applying an update operation (i.e., $u_{i,j} \in \mathcal{U}_i = (op_{i,j}, r_{i,j}, b_{i,j})$). In this way, application engineers always know whether they have to perform additional work, e.g., testing updated products.

Finally, the type of a guidance element (t_i in Table 7.1) specifies the automation degree of the guidance and can be *automatic*, *semi-automatic*, or *manual*. Automatic guidance can be applied fully automatic without the need for manual effort by application engineers. Semi-automatic guidance requires application engineers to choose between multiple possible update configuration operations that can be applied automatically. Manual guidance does not specify configuration update operations and application engineers have to adapt configurations on their own using the rationale of the SPL evolution. Manual typed guidance is required in case of domain engineers not being able to



(a) For domain engineers.

(b) For application engineers.

Figure 7.4.: Guided configuration evolution processes.

define update configuration operations. Guidance $\mathcal{G} = (r, \mathcal{X})$ is defined as a tuple containing the evolution rationale and the set of guidance elements. For instance, the exemplary guidance defined in Table 7.1 is specified for an evolution operation that deletes a feature f_0. Two guidance elements (x_1, x_2) are defined for configuration subsets not selecting f_0 (S_1) and selecting f_0 (S_2). Configurations not selecting f_0 can remain as-is $(op_{1,1})$ which preserves behavior $(b_{1,1})$ as the configurations are not affected by the evolution $(r_{1,1})$, and this configuration update can be applied automatically (t_1).

7.2.2. Guided Configuration Evolution Process

We propose processes for domain engineers to define guidance and for application engineers to apply such guidance. Figure 7.4a illustrates the process from the domain engineers' perspective. First, they perform SPL evolution independently of our method. This provides a high level of flexibility as we do not limit how to change an SPL. Optimally, domain engineers define guidance directly after performing SPL evolution, such that no details are forgotten. Domain engineers have to specify an evolution rationale which describes the essence of the performed SPL evolution. The rationale should be formulated such that application engineers with different levels of expertise and experience are able to understand it. Second, domain engineers have to determine configuration subsets for which they can define common configuration update operations. This is done by analyzing which features are involved in the evolution and which configuration subsets select them. For instance, if the feature `Encfs` of Figure 7.1 is deleted during evolution, all

configurations selecting `Encfs` are in one subset and all configurations not selecting `Encfs` are in another subset. Third, one or multiple update operations should be defined for each configuration subset and rationales explaining the update operations with their impact on product behavior must be added. Multiple update operations are necessary if the domain engineer identifies several sensible possibilities to update those configurations, for instance, if product behavior cannot be preserved or if it could be sensible to select new configuration options. However, if domain engineers are not able or do not want to define update configuration operations, respective update operations can be omitted. As a result, application engineers have to update their configurations on their own using the evolution rationale.

Fourth, domain engineers have to analyze how each configuration update operation affects product behavior to specify respective statements. This is optimally done with tool support, e.g., with a verification system that compares resulting artifacts of the original configurations before evolution with the update configurations after evolution. Domain engineers may define different levels of product behavior assurance. For instance, product behavior preservation is *proven* if it is shown using a proof system, whic is the highest level; product behavior preservation is *tested* if intensive testing resulted in the same product behavior; *reviewed* product behavior preservation is the lowest level and can be set if experts reviewed the resulting product and confirm product behavior preservation. These assurance levels are just examples and can be extended if necessary. Fifth, a guidance type determining the automation degree has to be set for each update operation. Domain engineers set the type to *automatic* if the update operation is without alternative, e.g., if the evolution was a refactoring or if only one configuration update operation is possible. Guidance should be *automatic* only if product behavior is preserved or if other circumstances force this operation (e.g., management decisions). If domain engineers defined multiple configuration update operations or if domain engineers are not sure whether the defined update operation is suitable, they set the type to *semi-automatic*. As application engineers are left without concrete suggestions on how to update their configurations, we consider it as undisciplined usage if domain engineers do not define update operations, and the type is set to *manual*.

Figure 7.4b shows the process from the application engineers' perspective. The knowledge transfer takes place when application engineers want to update a configuration to a new product-line version and consolidate the provided guidance. The concrete process differs based on the type of the provided guidance. For *automatic* guidance, the respective update operation can be applied automatically, without the need for manual effort from application engineers. Nevertheless, application engineers might be interested in how and why the configurations changes to potentially define own update operations. Thus, the update operation and the rationale can be investigated and automatic application can be canceled. For *semi-automatic* guidance, application engineers have to select the configuration update operation that best suits their needs based on the rationales. The selected operation can be applied automatically. However, if the update operations do not fully meet the application engineer's needs, the update operations can be adapted. For *manual* guidance, the application engineers can read the rationales that explain the SPL evolution, but have to find a configuration update operation on their own.

7.3. Guidance Templates

Specifying guidance for product-line evolution requires up-front effort which pays off if many application engineers update their configurations. To reduce effort for domain engineers, we provide a concept to store guidance for evolution scenarios in the form of templates to facilitate reuse. Consequently, guidance templates further automate the presented process, but are not necessary to apply our method. In contrast to guidance without templates, templates additionally specify the evolution scenario for which they are applicable. An evolution scenario $\mathcal{E} = (e_{\mathcal{F}}, e_{\mathcal{M}})$ consists of feature-model evolution ($e_{\mathcal{F}}$) and feature-artifact mapping evolution ($e_{\mathcal{M}}$), described in terms of the evolution operations we defined in Section 7.1. We deliberately do not consider the evolution of implementation artifacts $\mathcal{I}$ to $\mathcal{I}'$ and abstract from changes to implementation artifacts to be independent of a particular implementation language. Consequently, we just assume that $\mathcal{I}'$ is given after evolution. The evolution operations are preconditions for applying the guidance defined in the templates. Thus, an evolution template $\mathcal{T} = (\mathcal{G}, \mathcal{E})$ consists of a description of the evolution scenario and the corresponding guidance.

We define three exemplary guidance templates in this thesis for common evolution scenarios. We chose scenarios which related work identified as relevant evolution cases [PTD+16, SBT16, NBA+15, NSS16]. The templates also illustrate the general concept of guided configuration evolution. For brevity, we omit the rationales in the tables describing the templates but explain them in the text. To better reference elements of the table in the text, we add identifiers for guidance elements and update operations.

7.3.1. Delete Feature

Maintaining features and their mapped realization artifacts is expensive. Thus, it may not be profitable anymore to maintain certain features. In the running example (cf. Figure 7.1), the file system `Encfs` is outdated and rarely selected. Therefore, this feature is deleted, including its mapped artifacts. For such cases, we introduce the *Delete Feature* template.

In Section 7.2.1, we use this template to illustrate the structure of guided configuration evolution. Table 7.1 shows the elements of this template. As precondition, i.e., the SPL evolution that had to be performed in order to apply this template, the feature f_0 is removed and in the feature-artifact mapping application conditions, it is replaced by *false* as the feature can never be present anymore. For the configuration containing all configurations that do *not* select f_0, we define the first guidance element $Delete_0$. We specify the guidance category as *automatic* because such configurations remain unchanged, as they are unaffected by the operation, and explain this in the rationale.

We define a second guidance element $Delete_1$ for the configuration subset selecting f_0. As configuration update operation, we specify to remove f_0 and state in the update rationale that f_0 no longer exists. However, this does not preserve product behavior if artifacts were mapped to f_0 before evolution. Consequently, we set the guidance category to *semi-automatic* as application engineers should be informed of the reduced functionality. If application engineers decide for this update operation, it can be applied automatically.

7.3.2. Merge Features

Over the course of time, it may not be sensible to have multiple features that provide similar functionality as separately configurable features [PTD+16]. In our running example, the IP protocol ver-

Table 7.2.: Template *Merge Features*

Operation: Merge functionality of feature f_1 into feature f_0

$\mathcal{F}' = \mathcal{F} \setminus \{f_1\}, \mathcal{M}' = \mathcal{M}[f_1 \mapsto f_0]$

Configuration Subsets	Update Operations	Preserves Behavior	Type
$Merge_0 : c \in \mathcal{F} \upharpoonright (\neg f_0 \wedge \neg f_1)$	$c' = c$	yes	autom.
$Merge_1 : c \in \mathcal{F} \upharpoonright (f_0 \wedge f_1)$	$c' = c \setminus \{f_1\}$	yes	autom.
$Merge_2 : c \in \mathcal{F} \upharpoonright (f_0 \wedge \neg f_1)$	$M_{2.a} : c' = c$	no	semi-autom.
	$M_{2.b} : c' = c \setminus \{f_0\}$		
$Merge_3 : c \in \mathcal{F} \upharpoonright (\neg f_0 \wedge f_1)$	$M_{3.a} : c' = c \setminus \{f_1\}$	no	semi-autom.
	$M_{3.b} : c' = (c \cup \{f_0\}) \setminus \{f_1\}$		

sions `v4` and `v6` are required by both systems as `v6` is the new standard and many networks already rely on it but many other networks still require `v4`. As a result, in the configurations of most systems, both protocol versions are selected. Consequently, it would lead to unnecessary costs and effort to maintain both feature separately, as each feature has to work on its own and both in combination as well. For such cases, we define the *Merge Features* template which results in one feature providing the functionality of both original features. This reduces maintenance costs as it must not be ensured that each feature on its own must work.

Table 7.2 shows this template. The *source* feature f_1 is merged into the *target* feature f_0 and, thus, f_1 is removed from the set of features and f_1 is replaced by f_0 in all feature-artifact mapping application conditions.

We define four guidance elements for this template. The first element $Merge_0$ is for the configuration subset selecting neither f_0 nor f_1. The merge operation does not affect these configurations and, thus, we leave the configurations unchanged. As the product behavior is preserved, we set the guidance category to *automatic*. The second guidance element $Merge_1$ for the configuration subset which selects both f_0 and f_1. After evolution, the functionality of both features is provided by f_0 and, thus, we define as configuration update operation to remove f_1. As f_0 is still contained in these configurations, this results in preserved product behavior. Consequently, we set the guidance category to *automatic*.

We define third guidance element $Merge_2$ for configurations containing f_0 but not f_1. Existing approaches mainly fix invalid configurations [WSB+08, XHS+12, WPX+13]. As f_0 still exists, these approaches would leave these configurations as-is. However, as f_0 also provides the functionality of f_1, we know that products generated from these configurations *do not preserve* behavior. As it is not possible anymore to only select the functionality of f_0 before evolution without having the functionality of f_1 as well, it is not possible to preserve product behavior – even with another configuration update operation. In the third guidance element, we define two configuration update operations. Similar to other approaches, we define in the first configuration update operation $M_{2.a}$ that

Table 7.3.: Template *Extract New Feature*

Operation: Extract some functionality of feature f_0 into a new feature f_1

$\mathcal{F}' = \mathcal{F} \cup \{f_1\}, \mathcal{M}' \subseteq \{m' = m, m' = m[f_0 \mapsto f_1], m' = m[f_0 \mapsto (f_0 \wedge f_1)], m' = m[f_0 \mapsto (f_0 \vee f_1)] \mid m \in \mathcal{M}\}$

Configuration Subsets	Update Operations	Preserves Behavior	Type
$Extract_0 : c \in \mathcal{F} \upharpoonright \neg f_0$	$E_{0.a} : c' = c$	yes	semi-autom.
	$E_{0.b} : c' = c \cup \{f_0\}$	no	
	$E_{0.c} : c' = c \cup \{f_1\}$		
$Extract_1 : c \in \mathcal{F} \upharpoonright f_0$	$E_{1.a} : c' = c \cup \{f_1\}$	yes	semi-autom.
	$E_{1.b} : c' = c$	no	
	$E_{1.c} : c' = c \setminus \{f_0\} \cup \{f_1\}$		

the configuration is left as-is. In contrast to other existing approaches, we make application engineers aware that the product behavior is not preserved after applying this configuration update operation but that the functionality of f_1 will be contained additionally. Without this knowledge, products with altered behavior might be deployed which may cause harm. The second configuration update operation $M_{2.b}$ removes f_0 from configurations. Application engineers can select this update operation if they do not want to have the additional functionality of f_1 in the products and accept the loss of the functionality of f_0. As we do not know which configuration update operation is most suitable for application engineers, the guidance type is *semi-automatic*. Thus, application engineers must select an update operation that can be applied automatically.

$Merge_3$ describes the remaining case, i.e., for configurations that do not select f_0 but select f_1 and is defined similarly to $Merge_2$. As f_1 does not exist anymore after evolution, it must be removed from all configurations selecting it. We define two configuration update operations for this case. $M_{3.a}$ removes f_1 from the respective configurations. As this results in loss of f_1's functionality, we define that product behavior is not preserved. In the second configuration update operation $M_{3.b}$, f_0 is removed as well but f_1 is added. As a result, products of respective configurations provide the functionality of both, f_0 and f_1, after evolution, similar to $M_{2.a}$. Consequently, product behavior is not preserved as it additionally contains the original functionality of f_0. Again, we set the guidance type to *semi-automatic* as product behavior cannot be preserved and application engineers have to decide whether they want to lose the functionality of f_1 or can accept the additional functionality of f_0.

7.3.3. Extract New Feature

To allow more precise configuration with more configuration options, parts of a feature's functionality can be extracted into a separate feature. In our running example, the feature feature `Encfs` contains as a default the cryptographic cipher AES. To enable the introduction of more cryptographic ciphers which can be selected instead of AES, the functionality for this cipher is extracted

into a new feature AES. For such cases, we introduce the *Extract New Feature* template shifting functionality from a *source* feature into a new *target* feature.

Table 7.3 shows this guidance template. We add a new feature f_1 to the feature set. As some artifacts mapped to f_0 should be extracted to f_1, we need to represent this in the feature-artifact mapping. We identified four cases: first, if an artifact remains mapped to f_0 after evolution, we leave the feature-artifact mapping as-is; second, if an artifact belongs to the functionality that is extracted, we replace f_0 by f_1 in the application condition; third, if an artifact is required only to make both features work together, we replace f_0 by $f_0 \wedge f_1$ in the application condition; fourth, if an artifact is required by both features individually, we replace f_0 by $f_0 \vee f_1$ in the application condition. As the required evolution operation may differ for each artifact, product-line engineers can change each application condition independently.

We define two guidance elements. The first guidance element, $Extract_0$, targets configurations not selecting f_0. Principally, those configurations could be left as-is and product behavior would be preserved. However, application engineers potentially did not select f_0 before evolution as the entire functionality of f_0 did not match the requirements of that configuration. After evolution, new configuration options are available and application engineers might want to use them as they do not contain the entire functionality of f_0 before evolution. Consequently, we define three update operations. The first update operation $\mathcal{E}_{0.a}$ leaves corresponding configurations unchanged and preserves product behavior. The update operations $\mathcal{E}_{0.b}$ and $\mathcal{E}_{0.c}$ add f_0 or f_1 respectively, to the configuration. The two latter update operations do not preserve product behavior. To make application engineers aware of these new configuration options, we set the guidance type to *semi-automatic*.

The second guidance element $Extract_1$ targets subsets of configurations that select f_0. Again, product behavior could be preserved by adding f_1 to these configurations as the sum of the functionality of f_0 and f_1 matches the functionality of f_0 before evolution. Similar to $Extract_0$, application engineers might want to use the new configuration options. Consequently, we define three update operations. The first operation $\mathcal{E}_{1.a}$ adds the feature f_1 to the configurations as described above. The second operation $\mathcal{E}_{1.b}$ leaves the configuration as-is. The resulting product's functionality is reduced by the extracted functionality of f_1. The third operation $\mathcal{E}_{1.c}$ is relevant only if the functionality that has been extracted should be contained in a product. Correspondingly, f_0 is replaced by f_1 in configurations. Again, the latter two operations result in altered product behavior.

For this evolution scenario, existing approaches fixing defects in configurations [WSB+08, XHS+12, WPX+13] would leave the configuration as-is because f_0 still exists. In configurations covered by $Extract_0$, this would even preserve product behavior, but application engineers would not be informed about the new configuration options. However, in configurations covered by $Extract_1$ this would even lead to changed product behavior which may entail significant risk and cost to later fix and update these configurations.

7.3.4. Evolution Process with Templates

The three presented templates are examples that illustrate the usage of guided configuration evolution, and we do not claim completeness. For real-world SPL evolution, additional templates may be necessary. To this end, we enable domain engineers to define their own templates. However, as it is unlikely that a complete set of templates can ever be reached, our methodology can also be applied without templates following the process defined in Section 7.2.2.

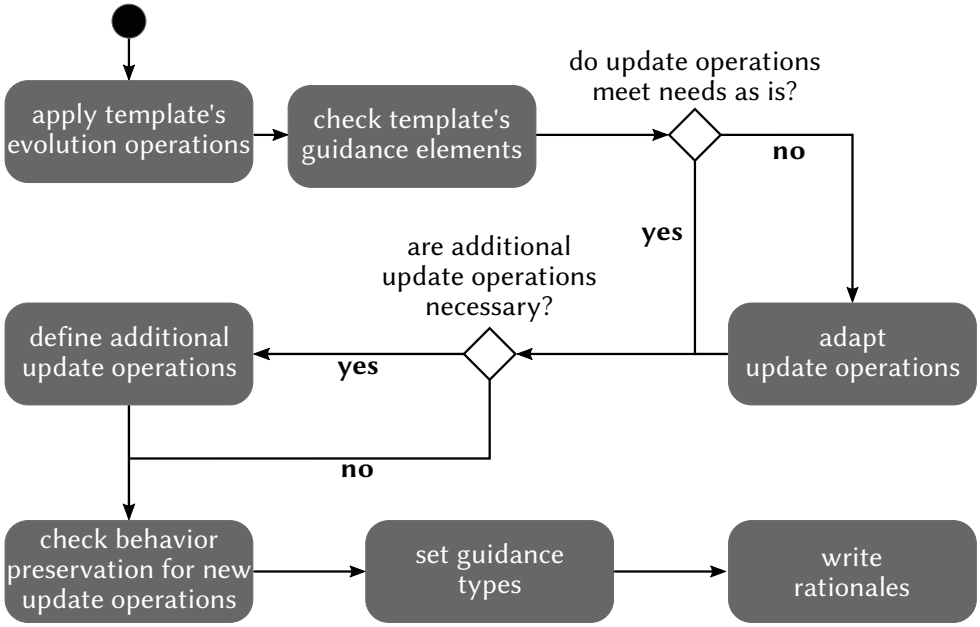


Figure 7.5.: Guided configuration evolution process for domain engineers using templates.

To cover the guided configuration evolution with and without templates, we need to adapt the process defined in Section 7.3.4. Figure 7.5 shows the process for applying a template from domain engineers' perspective. After selecting a template to be applied, domain engineers apply the defined evolution operations for the feature model and the feature-artifact mapping. Using suitable tooling, this step can be further automated. Two possibilities are: First, start guidance prior to SPL evolution and tools can semi-automatically apply evolution operations of a template. For instance, when applying the *delete feature* templates, domain engineers would only have to select which feature to delete and the tool modifies the feature model and the mapping accordingly. Second, after the fact, tools can help to identify suitable templates from a catalog of templates. As the feature-model and feature-artifact mapping evolution operations defined in the templates are preconditions for applying the template, template's operations have to match the actually performed changes to the SPL.

In the following steps, domain engineers have to check whether the elements defined in the template meet their needs. In an optimal scenario, the update operations meet the needs as-is and no modifications are necessary. However, domain engineers should always check whether they can define additional domain-specific update operations to better guide application engineers for a concrete evolution scenario. This would put application engineers in the position to make better decisions on which configuration update operation to apply.

If the update operations are not completely matching the evolution scenario or the intended way to update configurations, existing update operations can be adapted or can be supplemented by additional operations. For instance, if a feature should be replaced by another feature, the delete feature template can be applied with the first feature to be deleted, but domain engineers can adapt the update operations such that the first feature is replaced by the latter feature in configurations. For update operations, domain engineers need to analyze whether product behavior is preserved. If domain engineers claim product behavior preservation for new update operations, they have to ensure that the resulting artifact set is the same after the update, e.g., with a formal proof or exces-

sive testing. In the next step, the guidance types of the guidance elements should be set. We deliberately define this as a mandatory step to stimulate domain engineers in providing more information. Finally, the rationales for the evolution operation and the update operations must be written. This is of particular importance as application engineers should use this information as main source for decision making on how to update configurations.

By using templates, we expect that the effort for domain engineers to define guidance can be reduced. If a template can be used as-is, the effort is almost non-existent. Adapted or newly defined templates can be added to a template catalog and, over the entire life cycle of an SPL, the template catalog can grow to cover most evolution scenarios. Additionally, as the SPL evolution is semi-formally specified in the templates, our methodology lays the foundation for an automated detection of evolution scenarios and, thus, suitable templates. Such an automated detection would reduce the effort for domain engineers even more as they do not have to search for applicable templates. Even if effort for domain engineers remains unchanged, it results in proactively avoiding errors in configurations of application engineers instead of retroactively fixing errors. This process shows the flexibility of the guided configuration evolution as it can be used from scratch without any templates, it can be gradually extended by templates, existing templates can be reused directly, or existing templates can be adapted for a concrete scenario.

7.4. Proving Behavior Preservation

We formalized proofs for behavior preservation of the three templates using the theorem prover *PVS* [ORS92]. For the sake of brevity, we provide sketches of those proofs in the following. The complete proofs can be found in our online repository[2]. We use the formalization that we introduced in Section 7.1, i.e., product behavior of a configuration c is preserved by C', if $[\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_{c'}$.

For the *Delete Feature* template, behavior is preserved for configurations that did not select the deleted feature f_0 (cf. Table 7.1, $Delete_0$). To show this, we prove the following theorem.

Theorem 7.1: Behavior Preservation for Delete Feature Template Update Operations

For SPL $(FM, \mathcal{I}, \mathcal{M})$ evolved to $(FM', \mathcal{I}', \mathcal{M}')$, given that $\mathcal{I} \subseteq \mathcal{I}'$, $f \in \mathcal{F}, \mathcal{F}' = \mathcal{F} \setminus \{f\}$ and $\mathcal{M}' = \mathcal{M}[f \mapsto \mathit{false}]$:

$$\forall c \in \mathcal{F} \restriction (\neg f) : [\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_c$$

The idea of the proof is that we can show for an arbitrary $\mathcal{M}$, an $\mathcal{M}'$ exists for which $[\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_{c'}$, i.e., results in the same set of artifacts. We used *PVS* to provide this. In particular, we provde this by induction over the application conditions of all feature-artifact mapping entries. We have proven all of the following theorems in *PVS* using similar reasoning.

For the *Merge Features* template, behavior is preserved if either both features f_0 and f_1 were not selected in c or both features were selected (cf. Table 7.2, $Merge_0$ and $Merge_1$). In the first case, the configuration remains as-is and, in the second case, f_1 is removed from the configuration. To show behavior preservation for $Merge_0$ and $Merge_1$, we have proven the following theorem in *PVS*:

[2] https://gitlab.com/mnieke/guided-config-evo-eval-data

Theorem 7.2: Behavior Preservation for Merge Features Template Update Operations

For SPL $(FM, \mathcal{I}, \mathcal{M})$ evolved to $(FM', \mathcal{I}', \mathcal{M}')$, given that $\mathcal{I} \subseteq \mathcal{I}'$, with $f_0, f_1 \in \mathcal{F}, f_0 \neq f_1, \mathcal{F}' = \mathcal{F} \setminus \{f_1\}$, and $\mathcal{M}' = \mathcal{M}[f_1 \mapsto f_0]$:

$$(\forall c \in \mathcal{F} \upharpoonright (\neg f_0 \wedge \neg f_1) : c' = c, [\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_{c'}) \wedge$$
$$(\forall c \in \mathcal{F} \upharpoonright (f_0 \wedge f_1) : c' = c \setminus \{f_1\}, [\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_{c'})$$

In general, we are able to preserve product behavior for all possible configurations after applying the *extract new feature* template. We do not change configurations that do not select the feature f_0, and we additionally select the extracted feature f_1 for configurations that select f_0 (cf. Table 7.3, $Extract_{0.a}$ and $Extract_{1.a}$). We formalized and proved behavior preservation of the *extract new feature* template in *PVS* using the following theorem:

Theorem 7.3: Behavior Preservation for Extract New Feature Template Update Operations

For SPL $(FM, \mathcal{I}, \mathcal{M})$ evolved to $(FM', \mathcal{I}', \mathcal{M}')$, given that $\mathcal{I} \subseteq \mathcal{I}'$, with $f_0 \in \mathcal{F}, f_0 \neq f_1, \mathcal{F}' = \mathcal{F} \cup \{f_1\}$ and $\mathcal{M}' \subseteq \{m' = m, m' = m[f_0 \mapsto f_1], m' = m[f_0 \mapsto (f_0 \wedge f_1)], m' = m[f_0 \mapsto (f_0 \vee f_1)] \mid m \in \mathcal{M}\}$:

$$(\forall c \in \mathcal{F} \upharpoonright (\neg f_0) : c' = c, [\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_{c'}) \wedge$$
$$(\forall c \in \mathcal{F} \upharpoonright (f_0) : [\![\mathcal{M}]\!]_c = [\![\mathcal{M}']\!]_{c'})$$

To successfully apply our method, the templates and, especially, the feature-model and feature-artifact mapping evolution operations need to be applied correctly. This can be ensured by additional tool support that either applies the evolution operations automatically or verifies whether the performed evolution matches the defined evolution operations of a template.

7.5. Applying Guided Configuration Evolution

Guided configuration evolution is an extensive methodology and, thus, tool support is crucial for applicability in real-world development projects. Thus, we sketch the core functionalities a production tool needs to provide based on our methodology along with a suitable the workflows to realize the processes of Figures 7.4a, 7.4b, and 7.5. We implemented an early open-source prototype, named *GuyDance*, that provides some of these functions to show feasibility of our methodology (cf. Footnote 1).

Preserving compatibility with existing processes and tools is crucial for acceptance of our methodology. For this reason, we do not prescribe any tools to perform changes to an SPL, but domain engineers can perform SPL evolution with tools they are used to. This is of particular importance as guided configuration evolution can be used for important evolution operations, but does not have to be used for all operations. Domain engineers can decide from case to case whether they want to apply our method or not. Thus, if domain engineers consider a change as insignificant, our method does not have to be applied. For other more important changes, our method can be applied for other changes or it can be applied even retroactively when first problems occur.

For deep integration of our methodology, it may be sensible to provide tool support for semi-automatic application of template evolution operations. As the template evolution operations are

machine-readable, domain engineers could select which template should be applied to which feature, e.g., which features should be merged, and the tool can apply the respective changes to the feature model and the feature-artifact mapping automatically. For a more light-weight integration, further tool support can be sensible nevertheless. To increase the level of automation, templates that match the changes performed by domain engineers could be automatically detected by analyzing the performed changes and comparing to the changes defined in the templates. For instance, the tool FEVER [DDP17a] is able to extract and detect changes that match a certain evolution pattern, such as evolution scenarios described in templates. We show general feasibility of such an approach in our evaluation in which we detect changes to the Linux kernel feature model that match our templates using FEVER (cf. Section 7.6.2).

Next, domain engineers have to define guidance. If they applied a template, they can reuse the template's guidance operations with potential adaptations. If they did not apply a template, they must define new guidance. In the latter case and if domain engineers modified a template, they can save the new guidance as a new template. To define guidance and respective templates, a domain-specific language that provides the possibility to specify respective information is most suitable. In *GuyDance*, we used Xtext[3] for defining a grammar and editors for guidance and templates.

The first step for application engineers is to analyze which guidance elements (i.e., rows in the example tables) are relevant for an existing configuration. The configuration subset of a guidance element is formally defined and, thus, a suitable tool can automatically check whether a configuration is contained in a subset. For instance, if a subset is defined as $c \in \mathcal{F} \mid \neg f_0$ and a configuration selects features f_1, f_2, a SAT solver or a simple Boolean evaluation algorithm can check the formula $\neg f_0 \wedge f_1 \wedge f_2$ for satisfiability. In this example, the configuration would be part of the defined subset, i.e., that formula is satisfiable and the respective guidance element would be relevant. In the next step, a configuration update operation is selected for application. In case of an *automatic* guidance type, this can be done without interaction from application engineers. Nonetheless, a tool should provide the possibility for application engineers to inspect the update operation and the rationales. For *semi-automatic* guidance, application engineers have to select which update operation to apply. Thus, the different update operations with their rationales and the product behavior preservation statement should be listed. To increase user experience, the effect of these operations can be shown as a preview. After the selection of a configuration update operation, the execution can be fully automated. To apply an update operation, selected features of an existing configuration are either deselected or newly selected features are added to that configuration.

A specific characteristic of our methodology is its obliviousness of configuration validity. It is designed to use invalid configurations as input and may also take invalid configurations with it after applying guidance. Application engineers only needs to be consider configuration once after applying the guidance of potentially multiple evolution operations. This is particularly relevant as configurations are typically not directly updated to new SPL versions but skip multiple versions.

The approach to fix configuration validity only once after applying the guidance of all evolution operations has several benefits. First, configurations may be valid after applying the guidance of all evolution operations with respect to the final feature model after applying the last evolution operation. However, if configuration validity needs to be ensured after each evolution operation and intermediate evolution operations render configurations invalid, a fix would lead to a changed con-

[3] https://www.eclipse.org/Xtext/

figuration even though the original configuration would be valid after the last evolution operation. Second, application engineers may have to spend less effort if they consider configuration validity only once at the end as they do not have to consider it after each evolution operation. Third, fixing a configuration leads to divergence from the original configuration. Thus, if application engineers fix a configuration only once at the end, this may result in less divergence compared to fixing it multiple times for each evolution operation. In summary, guided configuration evolution is complementary to fixing configuration validity and to reestablish the validity of resulting configurations, we suggest to incorporate interactive configuration conflict resolution tools [TKS18b, WPX+13] once after applying the guidance of all evolution operations.

7.6. Evaluation

Our goal with guided configuration evolution is to *enable knowledge transfer from domain engineers to application engineers to update configurations after SPL evolution.* To show that we achieve this goal, we perform three complementary evaluations: first, we have proven behavior preservation in *PVS* (cf. Section 7.4) for typical evolution scenarios captured by our predefined templates; second, we perform a qualitative evaluation of our methodology based on the real-world evolution of our industry partner's SPL (cf. Section 7.6.1); third, we quantitatively evaluate for how many configurations we can preserve behavior or provide support that goes beyond state-of-the-art methods using the evolution of the Linux kernel SPL and real-world configurations of two product lines (cf. Section 7.6.2). In Chapter 6, we provide a concept to syntactically capture and plan evolution for arbitrary SPL artifact modeling languages. In this chapter, we provide a methodology to support the semantic part of updating configurations as part of SPL evolution. With the evaluation of this chapter, we want to contribute in answering **Research Question RQ3 – Consistent SPL Artifact Evolution How can we enable consistent evolution of SPL artifacts consisting of feature models, realization artifacts, feature-artifact mapping, and configurations?**. To this end, we pose the following research questions:

RQ3.4: Is it feasible to apply guided configuration evolution to real-world SPL evolution?

RQ3.5: To which extent does guided configuration evolution support updating configurations for real-world SPL evolution?

In our qualitative evaluation, we seek to answer **RQ3.4** and in our quantitative evaluation, we seek to answer **RQ3.5**.

7.6.1. Qualitative Evaluation

In the qualitative evaluation, we want to investigate the feasibility of guided configuration evolution (**RQ3.4**) by answering the following sub-research questions:

RQ3.4.1: Is it feasible to adapt guidance templates to fit real-world SPL evolution?

RQ3.4.2: Is it feasible to derive new guidance templates from real-world SPL evolution.

RQ3.4.3: What is the effort to define guidance for real-world SPL evolution?

Setup To retrieve realistic data on how SPL evolution would take place using our methodology, we interviewed our industry partner *Schnapptack* (`https://schnapptack.de/`) about the evolution of their web application SPL and problems emerging during this process. With their SPL, they

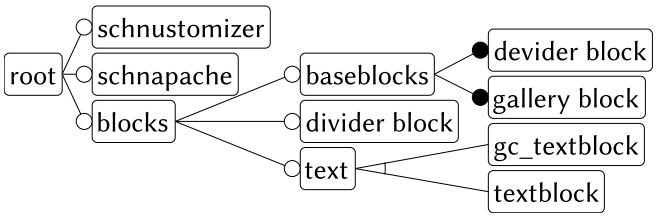


Figure 7.6.: Excerpt of Schnapptack's feature model.

can derive custom-tailored web applications that are based on different visual blocks of which the web application can be composed. The project is medium-scale with around ten developers working on it. We interviewed the project leader who is also involved in development activities. We asked our interview partner to describe which changes have recently been performed on the feature model and the feature-artifact mapping, and which changes are planned for the near future. Then we asked our interview partner about the reasons for the changes, how changes are related to each other, and how it is intended to update affected configurations. In total, we identified ten evolution scenarios that each contain related changes.

To provide support for these scenarios, we apply our pre-defined templates if possible. If the three example templates were not sufficient and, thus, could not be applied as-is, we adapted these templates to fit the scenarios to answer **RQ3.4.1**. If no existing template is suitable for a scenario, we define new guidance and, if possible, derive a new template from that scenario to answer **RQ3.4.2**. Finally, we investigated how much effort we spent to identify suitable templates, and adapt or change the templates to answer **RQ3.4.3**.

Figure 7.6 shows an excerpt of the feature model provided by *Schnapptack* before evolution. Most of the features represent blocks, which are visual components to build web applications. The original feature model contains 111 features, but we omit parts not affected by evolution. In the following, we explain five scenarios which we identified in the interview. The remaining five evolution scenarios were similar to the described ones, which results in the same insights.

Results *Scenario 1:* The feature `baseblocks` groups all basic block features (e.g., `devider block`, `gallery block`, ...). As the developers modeled each of the basic block features as MANDATORY, they decided to map all realization artifacts responsible for the data model to the feature `baseblocks`. To allow more fine-grained configuration options, all basic blocks become OPTIONAL, and the data models for each basic block feature are extracted from the feature `baseblocks` to the respective child features. Finally, `baseblocks` is deleted. To support this scenario, we adapt the *extract new feature* template. Instead of creating a new feature and feature-artifact mapping part of the functionality of an existing feature to the new feature, we now only change the feature-artifact mapping of part of the functionality to an already existing feature – without adapting the feature set. We perform this scenario for each of the sub-features. Then feature `baseblocks` is removed using the *delete feature* template.

Scenario 2: By mistake, two features implement the same functionality (`deviderblock` and `dividerblock`). Therefore, `deviderblock` is deleted together with its mapped artifacts. We capture this scenario by applying the *delete feature with mapped artifacts* template. As we know that both features implement the same functionality, we adapt the template's configuration update op-

erations so that it replaces `deviderblock` with `dividerblock` in configurations. This does not preserve product behavior as different artifacts are used, but we explain in the rationale that product behavior is similar. However, without our method, a repair operation would just deselect the feature `deviderblock` in configurations which would unexpectedly result in changed product behavior.

Scenario 3: The features `gc_textblock` and `textblock` implement similar functionality and share code which is mapped to the feature `text`. The `textblock` feature is the more mature feature, but the `gc_textblock` feature implements additional bug fixes. Thus, both features are merged into the feature `text`, and we apply the *merge feature* template twice. However, we adapted the template before applying it to `gc_textblock`. As only the bug fix should be integrated, we defined that only this part of the functionality should be mapped to the feature `text` after evolution. However, using the existing template twice, we were not able to represent this as one evolution operation. Hence, we defined a new template that merges multiple features into one feature.

Scenario 4: The feature `schnustomizer` provides certain functionality in a library that is used by other features. However, some features have a dependency to the feature `schnustomizer` only because of this library. Some other features are mapped to copies of that library. As a result, features that only use the library provided by the feature `schnustomizer` result in potentially unnecessary functionality in a product, this library is cloned multiple times in the SPL, and, even worse, the library exists in multiple versions. To resolve the unnecessary dependency to other functionality of `schnustomizer` and to resolve the redundancy of the library copies, a new feature should be created that is mapped to the library, and that is used by all features requiring the functionality of that library. The feature-artifact mapping of all other feature to (copies of) the library is removed.

We capture this scenario using the *extract new feature* template on feature `schnustomizer`. However, we adapt the template for configurations that select other features containing copies of the library before evolution. In particular, we define an additional guidance element for each configuration that selects at least one of those features. The configuration update operation of that guidance element is equal to $E_{0.c}$ of the *extract new feature* template (cf. Table 7.3), i.e., the extracted feature is selected as well, and we set the guidance category to automatic. Thus, the new feature is automatically selected in respective configurations without the need for interaction from application engineers. As configurations that originally select the feature `schnustomizer` or one of the other features using copies of the library might still be syntactically valid, e.g., if the usage of that library is optional, product behavior would change and without configuration guidance, application engineers might be unaware.

Scenario 5: In the last scenario, a webserver feature `nginx` is introduced to supersede the feature `schnapache`. The feature `nginx` is now the default webserver, but the `schnapache` webserver is still available. For this scenario, we create a new template containing two repair operations: first, to replace `schnapache` with `nginx` in all configurations; second, to leave each configuration as-is, i.e., continue using `schnapache`. As `schnapache` is still valid, we set the guidance category to *semi-automatic* and write in the rationale that we encourage using `nginx`, but that it is not compulsory. The first configuration update operation does not preserve product behavior but the second one does.

Discussion We were able to capture all evolution scenarios of *Schnapptack* and provide sensible repair operations. For Scenarios 1–4, we reuse our pre-defined templates but have to adapt the template configuration update operations or the feature-artifact mapping evolution operations to

fit the scenarios. For Scenario 2 in particular, we were able to simulate a *replace feature* operation by adapting a template, which shows the flexibility of our method. Thus, we can claim that it is feasible to adapt guidance templates to fit real-world SPL evolution (**RQ3.4.1**).

In Scenario 5, no existing template was fitting the requirements to recommend to replace a feature by new feature. This required to write the rationale, to identify the relevant configuration subsets (i.e., $c \in F \upharpoonright \mathit{Schnapache}$ and $c \in F \upharpoonright \neg \mathit{Schnapache}$), and to define configuration update operations for those configurations. In particular, we defined two possible update operations $c' = c$ and $c' = (c \setminus \{\mathit{Schnapache}\}) \cup \{\mathit{nginx}\}$ for configurations that select *Schnapache*. Configurations that do not select *Schnapache* can remain as-is, i.e., $c' = c$. We defined a new *replace feature* template based on this scenario and, thus, we are able to derive new guidance templates from real-world product line evolution (**RQ3.4.2**).

The effort we had to spend to define guidance (**RQ3.4.3**) slightly differs for each scenario. We had to adapt the existing *extract feature* template to support Scenario 1. In particular, instead of extracting functionality to a new feature, we used an existing feature as target. Thus, the only change to the template was to remove the feature-model evolution operation. Then, we applied the *delete feature* template which resulted in no additional effort as it already existed.

To replace the deleted feature in configurations by another feature in Scenario 2, we adapted the *delete feature* template. Thus, we only had to modify the configuration update operation correspondingly ($c' = (c \setminus \{\mathit{deviderblock}\}) \cup \{\mathit{dividerblock}\}$).

In Scenario 3, the templates could be applied as-is for one part of the evolution and, thus, this resulted in no additional effort. For the other part, we had to modify the feature-artifact mapping evolution operation to be applied only for one particular artifact. In Scenario 4, we adapted the *extract new feature* template and the main challenge was to identify the features with library copies which was done with support by our interview partner. In Scenario 5, we defined completely new guidance and a new template as described above.

In summary, we spent about 20 minutes per scenario to define guidance. Most effort was spent for identifying relevant features in Scenario 4 and for defining the entire guidance for Scenario 5. However, the latter took us about ten minutes which is little time compared to investigating multiple configurations individually.

Threats to Validity Internal validity of the qualitative evaluation might be biased as we interviewed only one person who may misinterpreted the evolution. However, this interview partner is a leading staff member with deep knowledge of the SPL implementation which reduces the chance for mistakes. Even if we misunderstood the SPL evolution, the evolution appears also to be plausible for other SPLs.

Another threat to internal validity is that we defined guidance by ourselves and measured the effort for it. On the one hand, we devised the guided configuration evolution methodology and, thus, have a high expertise in applying it. On the other hand, we are not at all involved in the development of the subject system. In summary, as they have domain knowledge, we expect that engineers involved in the development of the project require less effort to understand the evolution scenario and to devise sensible configuration update operations. In contrast, defining guidance most likely requires more effort by engineers.

The external validity is threatened as we only considered a medium-sized SPL project (111 features) and the evolution we analyzed is rather limited (10 affected features). However, this is neces-

sary as we needed to evaluate for each evolution operation how to apply guidance and whether this guidance is helpful. This requires manual effort and domain knowledge which is not possible for large data. Optimally, we would have accompanied the evolution of that SPL for a long period of time. However, this was not feasible due to our and our industry partner's limited resources.

7.6.2. Quantitative Evaluation

In our quantitative evaluation, we investigate to which extent guided configuration evolution supports updating configurations for real-world SPL evolution. We are mainly interested in the provided automation degree and the number of configurations for which we provide additional benefit compared to existing methods. To this end, we pose the following research questions:

RQ3.5.1: Which percentage of configurations can be supported by full automatic guidance?

RQ3.5.2: Which percentage of configurations requires knowledge of both domain engineers and application engineers?

RQ3.5.3: For which percentage of configurations can we preserve behavior after updating the configurations?

RQ3.5.4: Which percentage of configurations results in different product behavior after evolution that we detect but other methods would not detect?

Setup To provide meaningful results, we investigate the impact of hundreds of SPL evolution operations on thousands of configurations. As we need to consider evolution operations for which we already defined guidance, we consider evolution operations that match the templates of Section 7.3. As previous work identified these evolution operations as relevant [PTD+16, SBT16, NBA+15, NSS16], we use them as representative subset of possible evolution operations. For each occurrence of such an evolution operation, we analyze which of the defined guidance elements and update operations are applicable for the given set of configurations (i.e., the respective rows in Tables 7.1–7.3).

The quantitative evaluation is split into two parts that address the different roles involved in the guided configuration evolution process: first, we consider the domain engineer's perspective who performs SPL evolution but does not know existing configurations; second, we address the application engineer's perspective who knows about the configurations but was not involved in the SPL evolution. For the domain engineer's perspective, we use real-world SPL evolution and for the application engineer's perspective, we use real-world configurations. Our evaluation software and all data can be found in our online repository.[4]

Setup of Domain Engineer's Perspective As subject system with real-world SPL evolution, we use the Linux kernel. We search commits from its development history, using the tool FEVER [DDP17a], that match the evolution operations of the templates. For each of those commits, we extract the feature model before evolution using the tool KConfigReader [KGR+11]. We use the first six commits found by FEVER corresponding to each template. To this end, we analyze commits between Linux kernel versions 2.6.28 and 3.16. The analyzed feature model has between **8,003** (version 2.6.34) and **16,542** (version 3.16) features, depending on the kernel version and analyzed architecture (i.e., x86, ARM, MIPS, or PowerPC). Details on the commits can be found in our online repository.

[4] https://gitlab.com/mnieke/guided-config-evo-eval-data

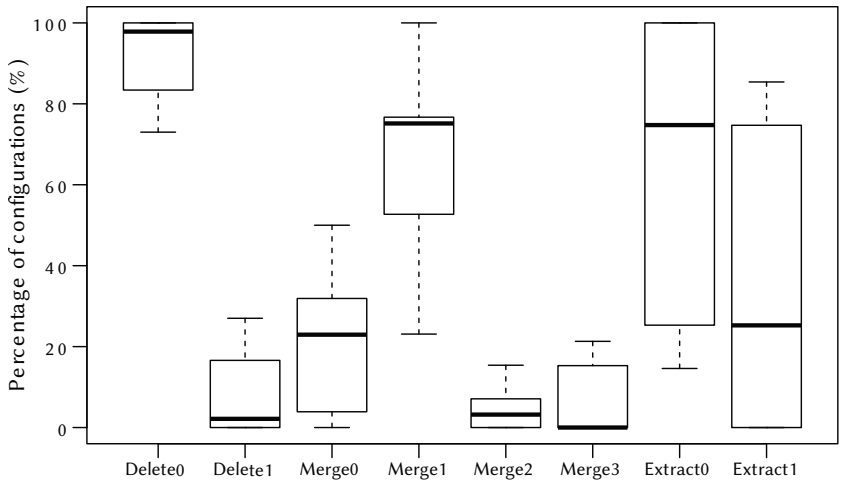


Figure 7.7.: Percentage of configurations covered by respective guidance elements for real-world evolution of the Linux kernel with six evolution operations for each template and 1,000 configurations.

Ideally, we would generate all possible configurations and analyze how guidance can be used for each of the configurations. However, it is not feasible to enumerate all configurations due to exponential growth of the number of configurations relative to the features [MR14b]. Therefore, and as domain engineers do not know which configurations are used in the field and why, we randomly generate configurations. In particular, we use the tool FeatureIDE in version 3.3 [MTS+17] to generate configurations. For each commit, we generate 1,000 valid configurations

Setup of Application Engineer's Perspective As subject systems for the evaluation from the application engineer's perspective, we use two real-world product lines and their real-world configurations [PMK+16, PSF+18, PSK+18]. The first product line *Agrib* consists of **2,008** features and **5,749** configurations. The second product line *ERP* consists of **1,728** features and **170** configurations. For both product lines, the evolution history is not accessible and, in contrast to the setup of domain engineer's perspective, we generate evolution operations and use real-world configurations. In particular, we generate multiple versions by randomly applying the evolution operations of our example templates. We generate 100 random operations for each template, resulting in 300 evolution operations for each product line. Although we randomly generate the evolution scenarios, we evaluate all available real-world configurations for each scenario. This setup matches application engineer's perspective as they do not know how and why an SPL evolved and, thus, SPL evolution appears random to them.

Results For both quantitative evaluations, we analyze for each occurred evolution operation how many configurations are covered by which guidance element of the templates. For instance, if an *extract new feature* operation occurred, we determine how many configurations are covered by the configuration subset of the guidance element $Extract_0$ or $Extract_1$ respectively. Therefore, we are able to determine for each configuration and evolution operation whether application engineers can automatically apply guidance and whether product behavior can be preserved.

Figures 7.7 and 7.8 show the aggregated results of the quantitative evaluation. Each data point of Figures 7.7 and 7.8 represents one occurrence of the respective evolution operation and shows the percentage of configurations covered by the respective guidance element. For instance, if 100 con-

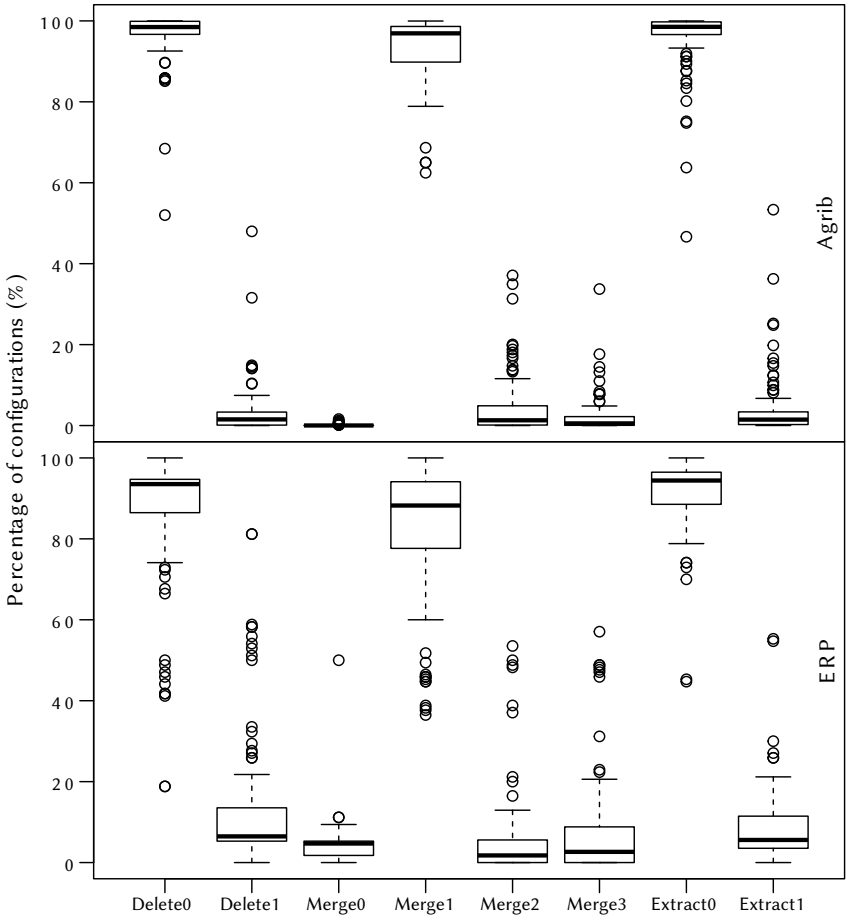


Figure 7.8.: Percentage of configurations covered by respective guidance elements for real-world configurations and 100 applications of each template evolution operation for the product lines *Agrib* (5,749 configurations) and *ERP* (170 configurations).

figurations exist, a *delete feature* evolution operation occurred and 80 configurations are covered by $Delete_0$, a data point at 80% for $Delete_0$ is added.

The results for both perspectives show similar patterns. For *delete feature* evolution operations, most configurations do not select the deleted feature and, thus, are covered by the guidance element $Delete_0$ (cf. Table 7.1). In the median, 97.9% of the configurations for the domain engineer's perspective and 95.2% of the configurations for the application engineer's perspective are covered by $Delete_0$. This is most likely the case because core features that are part of many configurations, i.e., on which many other features depend, are usually not deleted in real-world development. It is more likely that such features are becoming succesively unimportant and independent from other features, e.g., if the feature is replaced by a new feature. Finally, when nearly no other features depends on the considered feature, it is deleted. As we randomly selected features to delete in the evaluation from application engineer's perspective and only few configurations select the deleted feature, we expect that only few of such core features exist.

After applying a *merge features* evolution operation, most configurations are covered by the guidance element $Merge_1$ (in the median 75.2% for domain engineer's perspective and 92.9% for application engineer's perspective), i.e., these configurations contain both merged features. The sec-

ond most configurations for the domain engineer's perspective select none of the merged features and, thus, are covered by the guidance element $Merge_0$ (in the median 23.0%). This indicates that merged features are typically selected only in combination.

The results from domain engineer's perspective show a similar trend as the results from application engineer's perspective for *extract new feature* evolution operations. The main difference lies in the degree of scattering. In most configurations, the source feature is not selected and, thus, the configurations are covered by the guidance element $Extract_0$. For the domain engineer's perspective, the median value is at 74.8%, whereas for the application engineer's perspective, it is at 99.5%. The lower quartile of the configurations covered by the guidance element $Extract_0$ is at 31.9% whereas it is at 96.6% for *Agrib* and at 88.7% for *ERP*. This indicates that typically, no functionality is extracted from core features that are part of many configurations. However, the results for the domain engineer's perspective indicate that in real-world evolution, extracting functionality from a core feature is more likely than for applying the *extract new feature* to a randomly selected feature (cf. results from the application engineer's perspective).

Moreover, the results show that for some evolution operations, particular guidance elements cover no configurations, i.e., for $Delete_1$, $Merge_0$, $Merge_2$, $Merge_3$, and $Extract_1$. However, each guidance element still has its reason for existence at least for some evolution operations, they cover many configurations. For instance, only few configurations are covered by the guidance element $Merge_0$ for most *merge features* evolution operations, but in one case it covers **50**% of the configurations for one evolution operation for the *ERP* product line. Consequently, each guidance element is relevant as otherwise, situations exist in which for many configurations no guidance would exist.

Automation plays a pivotal role for guided configuration evolution. The higher the automation degree, the likelier that application engineers do not have problems when upgrading their configurations and, thus, they do not bother domain engineers. Thus, we analyze the automation degree (cf. **RQ3.5.1**). For guidance elements with *automatic* type, no additional effort by application engineers is required as their configurations are updated automatically. For guidance typed *semi-automatic*, application engineers only have to select the configuration update operation which fits best. For the *delete feature* template, we are able to automate guidance application ($Delete_0$) for between **18.8**% and **100**%. For the *merge features* template, we are able to automate ($Merge_0$ and $Merge_1$) between **43**% and **100**% of the cases. As new configuration options arise when applying the *extract new feature* operation, we deliberately do not provide any automated guidance to enable application engineers to select the new configuration options.

For *semi-automatic* guidance, knowledge of both engineer roles is required and, thus, application engineers need to become active when updating their configurations. Thus, for all configurations covered by the guidance elements $Delete_1$, $Merge_2$, $Merge_3$, $Extract_0$, and $Extract_1$, application engineers must select an update operation. To answer **RQ3.5.2**, knowledge of both engineers is required for few configurations in the median for the *delete feature* and *merge feature* templates. However, in the worst case, for **81.2**% of the configurations, knowledge of both engineers is required. For the *extract new feature* template, knowledge of both engineers is required for all configurations. The configurations for which knowledge of both engineers is required, are those configurations for which existing methods [SBT16, WSB+08, WPX+13, XHS+12, NSS16] do not suffice as they do not provide the possibility to share knowledge between the engineers. Even if the median values are

not high, in some cases for up to **81.2**% of the configurations, application engineers would be left alone in updating the configurations if they these existing methods.

One goal of updating configurations to a new SPL version is to preserve behavior of the resulting products. Even if application engineers want to modify their existing configurations, a configuration with the same behavior compared to before evolution suits best as a starting point. Thus, in **RQ3.5.3**, we are interested in the number of configurations for which product behavior can be preserved. The respective relevant guidance elements for the delete and merge templates are $Delete_0$, $Merge_0$, and $Merge_1$. This exactly matches the automation degree for those templates. Even if we do not provide full automated guidance for the *extract new feature* template, we are able to preserve product behavior for *all* configurations. Most of the configurations can remain as they were before evolution to preserve behavior, i.e., $Delete_0$ and $Merge_0$. Thus, without our method, behavior would be preserved as well but with our method, application engineers can be assured that they do not encounter unexpected behavior changes and do not have to check or test this themselves.

Most of the existing methods to update configurations only consider configuration validity and provide respective fixes [WSB+08, WPX+13, XHS+12, NSS16]. However, this can unexpectedly result in changed product behavior. Consequently, these are the most crucial cases in which application engineers might not be aware of the changed behavior and deploy respective products. This may result in severe problems and costs. With **RQ3.5.4**, we investigate in how many cases our method detects product behavior changes that would not have been detected using the existing methods to update configurations. In particular, configurations covered by the guidance elements $Merge_2$ and $Extract_1$ would result in potentially unnoticed product behavior changes. The median values for those guidance elements are not high but in the worst case, product behavior changes of more than half of the configurations (**55.3**% for one *extract new feature* evolution operation in the *ERP* product line) would not be detected. Even if only few configurations fall into those categories, these are critical cases that may lead to severe problems.

In summary, guided configuration evolution is relevant in many cases, especially if knowledge of both engineer roles is required. We are able to automate guidance to a high degree between **18.8%** and **100%**. This enables application engineers to update their configurations without spending a lot of effort. For the same percentage of configurations, product behavior can be preserved. Most configurations do not require knowledge of both engineers, but in some cases up to **81.2%** of the configurations require this knowledge combination. Without out methodology, no approach exists that enables this knowledge combination. Furthermore, we are able to detect behavior changes that would remain unnoticed using other methods for up to **55.3%** of the configurations. Thus, our methodology allows to provide a high degree of automation while frequently preserving product behavior and in the case of product behavior changes, we save application engineers from not perceiving this.

Threats to Validity The internal validity of our quantitative evaluation is threatened as we consider two extreme roles: the domain engineer who does not know anything about existing configurations and the application engineer who does not know anything about SPL evolution. In real-world projects, it is more likely that domain engineers are involved in maintaining at least some configurations and that application engineers know some core technical details about the SPL and its evolution. Consequently, no or less guidance is required for the previously described engineers when updating configurations, and those engineers might directly know whether product behavior is pre-

served or not. Applying our methodology would then result in higher non required effort. However, even if domain engineers maintain some configurations or application engineers know technical details, the time points of SPL evolution and configuration updating potentially lie weeks or months apart. In the meantime, engineers might have forgotten how the SPL evolved since the last configuration update and multiple evolution steps might exist. Our methodology serves also as detailed evolution documentation and engineers can come back to this documentation if they do not remember all details of the evolution anymore. Moreover, the more critical situation is when domain and application engineers are strictly separated and they cannot update configurations alone.

The random generation in our quantitative evaluation may also affect the internal validity. As we are not aware of any open-source product line for which commit history **and** existing configurations are publicly available, we decided to generate configurations for Linux and generate evolution scenarios for the *Agrib* and *ERP* product lines. Thus, part of the evaluation uses real-world SPL evolution and other parts use real-world configurations. Considering all configurations and all possible applications of templates is not feasible due to combinatorial explosion. However, this problem confirms our challenge: domain engineers do not know which configurations exist. To reduce the bias that may be introduced due to the random generation, we heavily used repetitions by considering 1,000 configurations for each commit of the Linux kernel and 100 applications per each template for both *Agrib* and *ERP* product lines. The random generation of configurations only affects the evaluation of the Linux kernel, but not the *Agrib* and *ERP* product lines. Random application of templates only affects the evaluation of the *Agrib* and *ERP* product lines but not the evaluation of the Linux kernel. As the results for both perspectives are similar, we expect these results to be representative. For the random configuration generation, we used FeatureIDE (version 3.3) which does not generate uniformly distributed configurations. However, tools/methods to generate uniformly distributed configurations do not (yet) scale for large variability models as used in the evaluation [OGB19]. Additionally, real-world configurations are not uniformly distributed, and it is not possible to make statements about the distribution without domain knowledge or real-world data.

The tools which we used for the quantitative evaluation may affect internal validity as they may contain defects. With KConfigReader and FEVER, we rely on tools that have been used in prior studies [KGR+11, DDP17a, SBT16, GTA+19, EKS15]. In particular, FEVER may detect too few or too many commits matching the templates in the history of the Linux kernel. It is uncritical if FEVER misses commits matching evolution operations in the Linux kernel history as we were interested neither in all commits matching the operations nor in the probability of occurrence. Furthermore, we manually inspected all detected evolution operation occurrences and could confirm that the respective evolution operation was detected correctly.

To reduce the threats to external validity, we use a combination of strategies. First, we analyzed a total of three real-world SPLs from different domains. Second, we analyzed one open-source and two closed-source SPLs with different implementation techniques. Thus, we reduced the threat that our method is applicable only to systems with a certain nature. Finally, we focused on evolution operations that have been identified as relevant in the literature [PTD+16, SBT16, NBA+15, NSS16] and which we indeed could confirm for Linux.

7.7. Related Work

Knowledge transfer is a wide-spread topic in software engineering. Jihong et al. [Jih10] classify knowledge transfer in *structured* and *unstructured* methods / approaches. Whereas our method is a structured knowledge transfer method, the authors highlight that unstructured knowledge transfer routinely takes place. However, a major reason for unstructured knowledge transfer is a communication barrier between employees. Guided configuration evolution reduces this communication barrier and, thus, increases structured knowledge transfer.

The evolution of highly configurable software systems has been subject to recent research. Xu et al. [XZH+13] identified misconfigurations that lead to vulnerabilities or bugs. In particular after system evolution, configurations are often not updated which may entail significant problems. Xu et al. conclude that developers should provide support users in the configurations process to fix misconfigurations. We address this issue with our methodology as we enable domain engineers (i.e., developers) to support application engineers (i.e., users).

Zhang et al. [ZE14] address a similar problem as guided configuration evolution. Their goal is to preserve product behavior after evolution by analyzing products' control flow behavior. As a result, they suggest configuration update operations that result in the most similar product behavior. However, Zhang et al. [ZE14] uses a white-box approach, but our approach is more conservative and a almost black-box approach. Moreover, they do not enable to deliberately change configurations, e.g., to replace features, is not possible. The method of Zhang et al. [ZE14] could be used complementarily by domain engineers for cases in which product behavior cannot be preserved to devise a suggestion for a update operation.

Classification of SPL Evolution Recent research analyzed and categorized evolution of SPLs [BKL+16, DDP17a, PTD+16, ZRL16]. Passos et al. [PTD+16] extracted evolution patterns from evolution of the Linux kernel variability model and associated artifacts. We used some of these templates as real-world feature model evolution scenarios for our templates.

Bürdek et al. [BKL+16] retroactively derive performed evolution operations by a differencing mechanism between two feature-model versions. The authors provide a comprehensive catalog of typical evolution operations for feature models. To derive evolution operations, they compute the differences between two feature-model versions and map them to the evolution operations. However, Bürdek et al. do not consider changes to the feature-artifact mapping and, thus, their method does not support making statements about product behavior.

With FEVER, Dintzner et al. introduced a tool to extract changes to variability models, code artifacts, and the corresponding feature-artifact mapping [DDP17a]. As highlighted in Section 7.5, FEVER could be used in combination with our methodology to identify commits of an SPL that match a certain pattern, such as the evolution scenarios described by guidance templates. We used FEVER to find and extract the commits of the Linux kernel for our quantitative evaluation (cf. Section 7.6.2).

In [PTD+16, BKL+16, DDP17a] commits are categorized, but the guided configuration evolution is more generic and helps to update configurations. Ziegler et al. analyze which changes of the Linux kernel variability model affect other artifacts [ZRL16]. The results are used for regression testing of configurations that are mapped to changed artifacts to explicitly test these changed artifacts. However, they do not provide support for fixing configurations. Their approach could be im-

proved incorporating product behavior preservation properties of evolution operations. Multiple authors identify dead or superfluous `#ifdef` blocks (i.e., feature-artifact mapping entries) [ZRL16, TLS+11, TLD+11, NDT+13]. Such analyses could be integrated with guided configuration evolution to check for new dead or superfluous feature-artifact mapping entries after each evolution. Respective entries can then be removed from the feature-artifact mapping.

Alves et al. [AGM+06] and Thüm et al. [TBK09] reason on feature-model evolution in terms of changes to the set of valid configurations. Alves et al. [AGM+06] define operations which result in a refactoring, i.e, no configurations are removed or added. Thüm et al. [TBK09] goes beyond and also defines categories for evolution operations that introduce or remove configurations or combinations thereof. Both approaches do not consider product behavior of configurations. Schulze et al. incoporate feature-artifact mappings to define refactoring operations for product lines using feature-oriented and delta-oriented programming [STK+12, SRS13]. Seidl et al. define typical evolution operations which can be used to co-evolve three spaces: feature models, artifacts, and feature-artifact mappings [SHA12]. If an evolution operation affects more than one space, they define how to co-evolve the other spaces. In contrast to the previously mentioned publications, we do not just categorize evolution operations and do not limit on refactorings.

Borba et al. introduced a refinement theory for SPL evolution preserving product behavior [BTG12]. Neves et al. proposed a set of evolution templates preserving product behavior using this theory [NBA+15]. Sampaio et al. extended this theory by the notion of partially safe evolution operations that preserve product behavior for a subset of configurations [SBT16]. The refinement and partial refinement theories enable to reason on configurations for which product behavior is preserved – even if configurations need to change. With guided configuration evolution, we devise a more general methodology that overcomes the communication barrier between domain and application engineers. Additionally and in contrast to the (partial) refinement theories [SBT19, SBT16, NBA+15, BTG12], guidance can also be specified if product behavior cannot be preserved. Our formalization and proofs base on the works of of Borba et al. [BTG12], Neves et al. [NBA+15], and Sampaio et al. [SBT19, SBT16].

Repairing Configurations Different research addresses fixing configurations that became invalid. White et al. [WSB+08] introduce an automatic approach that computes the smallest possible set of changes in a configuration to fix it. Xiong et al. [XHS+12] propose a semi-automatic approach to provide a complete set of possible fixes with the smallest number of feature changes. In contrast, Wang et al. [WPX+13] incorporate application engineers' feedback to gradually reach a desired fix. Both semi-automatic approaches assume that the person fixing the configuration knows what the best fix is. Moreover, all of these approaches do not consider the implementation and feature-artifact mapping. As a result, provided fixes may lead to different product behavior and, therefore, provide a false sense of correctness.

7.8. Chapter Summary

In Chapters 3 – 6, we provide methods to model the evolution of an SPL with a particular focus on paradox-free and anomaly-free feature models. In this chapter, we address **Challenge 5: Updating Configurations after SPL Evolution** regarding updating configurations after SPL evolution. To this end, we answer **RQ3.4** (applicability to real-world SPLs) and **RQ3.5** (benefit) which contribute in an-

swering **Research Question RQ3 – Consistent SPL Artifact Evolution**. We present guided configuration evolution, a methodology for updating configurations after SPL evolution that overcomes the communication barrier between domain engineers and application engineers. domain engineers can define guidance consisting of the essence of SPL evolution and recommended configuration update operations. Application engineers can use this guidance to automatically update their configurations if possible or to make an informed decision on how to update their configurations. As guidance defines whether product behavior is preserved if a certain configuration update operation is applied, application engineers are always aware of the consequences of applying particular update operations. Our methodology excels in situations in which it is impossible to communicate for domain engineers and application engineers. Application engineers can update their configurations in accordance with SPL evolution at the time of their choosing, and with the most suitable update strategy. domain engineers have to spend effort for defining guidance only once per evolution operation and this guidance can be used by an unlimited number of application engineers.

This work raises several further research opportunities. First and most importantly, we lay the theoretical and practical foundations for guided configuration evolution and show the relevance in our practical evaluation. To assess effectiveness, efficiency, and acceptance for real-world SPL evolution processes, we plan to perform a long-term study with our industry partners. As an extension to our methodology, we want to support compound templates or batches of templates which are applied as a cohesive sequence. During our qualitative evaluation (cf. Section 7.6.1), we defined a *replace feature* operation. This leads to the idea of not only defining guidance in case of feature-model or feature-artifact mapping evolution operations but also to update configurations based on new requirements or business needs but without evolution of the SPL itself. Moreover, we want to investigate automatic learning from modified templates (either by domain or by application engineers) to derive new templates or to sustainable change templates. A further future work opportunity is an extension of our method that ensures configuration validity after applying configuration update operations, which would reduce manual effort of application engineers even more. Finally, if domain engineers define their own templates, automatic proofs of behavior preservation would increase usability.

8 Conclusion

In this chapter, we conclude this thesis with a summary of our contributions, a discussion of our results, and an outlook on future research areas that are based on our contributions.

8.1. Contribution

In this thesis, we provide concepts to capture and plan consistent and anomaly-free SPL evolution. Figure 8.1 illustrates the contributions of this thesis and, in the following paragraphs, we summarize each of them, grouped by our research questions.

Research Question RQ1 – Modeling the Entire Feature-Model Evolution Timeline With TFMs, we introduce a concept to capture an entire feature-model evolution timeline within the same artifact. We achieve this by modeling each element of a feature model as a temporal element that has a temporal validity – an interval in which it is temporally valid. Consequently, the temporal relation between all evolution steps is directly stored in a TFM. This enables us to model future feature-model evolution while modifying the current state in parallel. Additionally, to refine or replan evolution, intermediate evolution steps between already planned evolution steps can be introduced as well. If plans are modeled using a TFM, they automatically become present and, afterwards, past history as time passes. In summary, TFMs can be used to plan and drive SPL evolution in its entirety. Our evaluation shows that we are able to capture real-world feature-model evolution using TFMs and our tool suite DarwinSPL.

Research Question RQ2 – Feature-Model Inconsistency and Anomaly Prevention We determine that feature-model inconsistencies that are introduced during evolution only occur if changes of a retroactively introduced intermediate evolution step conflict with changes of already modeled future evolution. We denote such an inconsistency as *evolution paradox*. To guarantee paradox-free TFMs, we define an execution semantics for feature-model evolution plans. If a new evolution operation is applied to a TFM, we check whether this operation would introduce a paradox and, in this case, we prohibit the evolution operation's execution. As an input, we use an initial state of the feature model and a sequence of evolution operations. This sequence consists of already planned evolution operations as well as the newly induced operation. For each evolution operation, we check whether any TFM well-formedness rule is violated. If such a rule is violated, an evolution paradox would be introduced. Thus, we are able to perform, plan, and *replan* feature-model evolution while preventing the introduction of evolution paradoxes. In our evaluation, we show that our method scales for large-scale real-world feature models and their evolution. Additionally, we empirically show that both researchers from academia as well as practitioners from industry consider the problem we address as very important and our method as valuable.

With our evolution-aware anomaly detection, we detect anomalies in evolution steps of a TFM. To this end, we encode the entire timeline of a TFM in one query for a solver. This enables us to reuse the solver for unchanged parts between different evolution steps, and to directly deter-

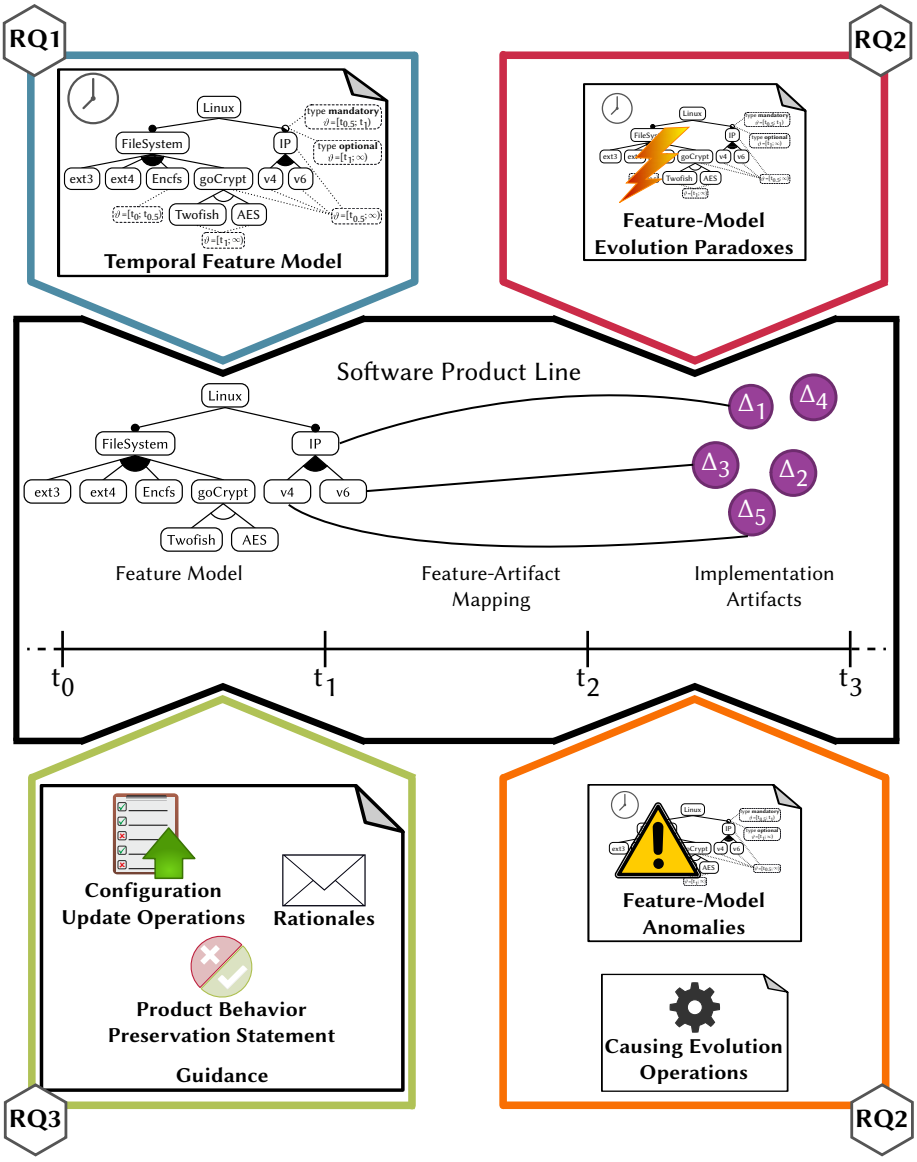


Figure 8.1.: Overview on our contributions.

mine the time point of anomaly introduction. To provide concise yet expressive anomaly explanations, we derive the causing evolution operations for each anomaly. In particular, we achieve this by linking TFM elements to clauses of the solver query. When explaining an anomaly using a solver, it returns unsatisfiable clauses which we can translate back to TFM elements. Thanks to the temporal validities and evolution operations stored in the TFM, we can derive which evolution operation occurred to the respective elements at the time point of anomaly introduction. These evolution operations are the immediate cause for an anomaly. In our evaluation, we show that we correctly detect all anomalies in real-world feature model evolution timelines and that our method scales for large-scale feature models. However, we are not able to show a performance advantage for encoding the entire TFM in one solver query. Additionally, we show that our method reduces anomaly explanation complexity significantly.

Research Question RQ3 – Consistent SPL Artifact Evolution To enable consistent SPL artifacts, we generalize the concepts of TFMs and temporal elements applied for arbitrary modeling languages, which forms a uniform language concept to model evolution. In particular, we provide a method that automatically generates an augmented metamodel based on a given metamodel. This augmented metamodel enables to store model evolution in terms of past history and planned evolution, similar to TFMs. As we know that compatibility with existing modeling and analyses tools is pivotal for acceptance of our method, our generation process preserves compatibility to the existing metamodel. In particular, we generate an adapter infrastructure, that uses an augmented metamodel as backend but can be used as if it was a non-augmented metamodel. When modifying a model instance using the non-augmented metamodel's interface, the changes are automatically tracked as evolution in the augmented model instance. Future applications or analyses can then be devised that make use of the augmented metamodel's capabilities. In summary, our method enables to augment arbitrary metamodels to store an entire model evolution timeline in one artifact. For SPLs, this may comprise feature models, implementation artifacts, documentation, feature-artifact mappings, or configurations. In our evaluation, we show that we are able to store real-world model evolution in augmented model instances and that our method scales to real-world metamodels. The metamodel augmentation can be used to model consistent evolution of SPL artifacts consisting of feature models, realization artifacts, feature-artifact mappings, and configurations. However, we do not provide methods that *ensure* consistency between those artifacts.

As a first step towards to ensure consistent evolution of SPL artifacts, we address the evolution of configurations in concert with feature models, realization artifacts, and feature-artifact mappings with our guided configuration evolution methodology. We overcome the communication barrier between domain and application engineers by enabling domain engineers to perform SPL evolution and then define guidance on how to update configurations. Such guidance consists of concrete update suggestions for configurations, rationales for the SPL evolution and the update operations, and statements whether product behavior will be preserved after evolution. Application engineers use this guidance to update their configurations. In some cases, this can be done automatically, whereas in other semi-automatic cases, application engineers need to decide for a suggested configuration update operation. Guidance is defined only once by domain engineers and can be used by an unlimited number of application engineers. Moreover, application engineers can update their configurations using guidance at a time of their choice. In summary, we enable knowledge transfer from domain engineers to application engineers such that application engineers are able to make informed decisions on how to update their configurations. Additionally, unintentional and unnoticed product behavior changes cannot occur using our methodology. In our evaluation, we show that our methodology is applicable to real-world SPL evolution, that we can automate most configuration updates, and that we cover a significant number of cases in which product behavior would be changed unnoticed.

All previously described contributions serve to answer our **Main research question: How can we track, execute, and plan feature-model evolution to drive consistent SPL evolution?** in its entirety. With TFMs, we enable modeling entire feature-model evolution timelines that serve as starting point for SPL evolution. Our analyses in Chapters 4 and 5 ensure that the feature-model evolution timeline itself is consistent. With the metamodel augmentation method (cf. Chapter 6), we provide the basis to consistently model evolution of all SPL artifacts, including feature models, re-

alization artifacts, and feature-artifact mappings. Thus, these artifacts can be kept in a compatible state. Finally, with guided configuration evolution (cf. Chapter 7), we provide a methodology to ensure consistency of configurations with feature models, realization artifacts, and feature-artifacts mappings. Even if we ensure consistency only of TFMs and configurations, we claim that our methods are an answer to our main research question and can be used to establish TFMs as driver for consistent SPL evolution including configurations.

8.2. Discussion

We made certain design decisions for each of the presented methods in this thesis. Consequently, our methods excel in certain situations while for other situations, potential for improvement may exist. In the following, we will discuss benefits as well as limitations of each of our contributions.

8.2.1. Temporal Feature Models

With TFMs, our goal was to enable capturing feature-model evolution in one artifact with a particular focus on planning and replanning. Additionally, we focused on structural evolution on element basis in form of temporal elements instead of focusing on evolution operations. As a result, we defined a complex, but expressive metamodel. Retrieving information regarding performed evolution operations requires additional computations. However, we decided to use this notation instead of modeling evolution operations in the first place as we are more flexible this way. In particular, evolution operations can be defined on the basis of temporal validities, and even more complex compound operations can be defined in the same way. As a result, tools performing or analyzing such complex operations can make use of extensions, whereas basic tools can still work on the basis of temporal elements and do not have to deal with complex operations.

In discussions about TFMs, we were often asked why we did not just implement the same using existing technologies such as Version Control Systems (VCSs). In fact, implementing a similar method using VCSs would have been easier, and engineers using such a tool would potentially be more used to the workflow. However, VCSs are in general language-agnostic and, thus, work on line basis of text files. While it would be easy to derive different feature-model versions using such an approach, it would be very complex, computationally expensive, and potentially inaccurate to compute the differences between feature-model versions. Thus, we decided for modeling temporal elements that directly store necessary information. Operation-based VCSs would provide a remedy, but we decided against using operations as basis to model evolution as described in the paragraph above. Moreover, replanning using VCSs would be very complicated as commits would have to be introduced between existing commits. As a consequence, the system would need to re-compute all subsequent commits when the basis changes and, thus, changes in these commits may have to be adapted.

8.2.2. Evolution Paradox Detection

Feature-model inconsistencies can occur in all kinds feature models. However, for standard feature models, it is simple to prevent the introduction of inconsistencies. For instance, if a feature is deleted, all of its sub-features need to be deleted as well as, otherwise, the sub-features would not have a parent feature. In TFMs, and especially in presence of *replanning*, evolution paradoxes are more complex inconsistencies that are hard to detect. One question that we discussed often is whether we created the basis for evolution paradoxes with TFMs in the first place. While it is true

that standard feature models do not have the problem of evolution paradoxes, as soon as planning and replanning of feature-model evolution is considered, methods like TFMs become irreplaceable. The additional expressiveness and capabilities of TFMs comes at the price of additional inconsistencies. Moreover, one of our industry partners plans feature-model evolution using multiple standard feature models. They also have evolution paradoxes across these feature models, but detecting and fixing them is significantly more complex than with TFMs as they have independent feature-model versions in different files.

When validating our execution semantics to detect potential evolution paradoxes and to prevent their introduction, we implemented a cross-check in Java to ensure that our execution semantics detects all evolution paradoxes as we expect it to be. When comparing computation times, it turns out that our Java implementation was always quicker for all of our tested subject systems. Thus, a legitimate question is whether we need the execution semantics at all. First of all, we were only able to implement the Java cross-check after we had defined formal semantics for feature-model evolution operations and evolution plans. Before defining the formal basis, we tried to implement a similar check in Java. However, we failed because of the complexity of many corner cases we needed to consider. With the formal foundation, we have a very concise definition feature-model consistency regarding evolution operations that is independent of a specific programming or feature-model language.

8.2.3. Anomaly Detection and Explanation

Our anomaly detection in feature-model timelines encodes the entire timeline in one query. As a result, the solver can reuse parts that stay the same for multiple evolution steps. While we expected that this would increase performance, this is not always the case. In particular for feature anomaly analyses (i.e., dead or false-optional features), it is even slower. This may be the case because the solver for analyzing individual feature model versions without the encoded timeline can already reuse nearly all formula clauses when switching between features to be analyzed. Moreover, the formula that contains the entire evolution timeline is larger than the formula for each individual version. Thus, it may depend on the use case whether using the encoded evolution timeline is faster or not. However, other factors such as optimizations in solver queries may play an additional role. For instance, it may be beneficial to order the clauses differently based on the evolution steps for which they are valid. Another possibility may be to implement incremental analyses by using results from analyses of previous feature-model versions. With TFMs, this becomes easier as no additional pre- or post-processing is necessary to define such solver queries or interpret the results in the light of multiple feature-model versions.

The explanations we provide for anomalies consist of evolution operations that engineers performed during evolution. Our claim is that evolution operations are easier to understand than clauses of a formula. While this might be true for most engineers, some engineers might more efficiently fix anomalies by using formula clauses. However, we still think that the evolution operations serve well as a default and that engineers who have a more theoretical background can additionally use the formula clauses. As explained in Chapter 5, in-depth studies with different user groups need to be conducted to understand which information can be used best to fix anomalies.

8.2.4. SPL Artifact Evolution

With our metamodel augmentation, we are able to store evolution of arbitrary SPL artifacts. However, this requires a metamodel that represents the language used to define all artifacts. Many artifacts are already defined in models that are based on a metamodel. However, especially programming languages are typically defined using grammars instead of metamodels. Consequently, they are not modeled and, thus, our metamodel augmentation is not directly applicable. Nevertheless, it is possible to define metamodels for such languages. For instance, in our evaluation, we showed that we are able to capture the evolution of Java programs using the JaMoPP metamodel. Nonetheless, Java programs are typically not developed using an editor based on the JaMoPP metamodel. Thus, compatibility with those grammar-based tool chains is not preserved when our augmentation method is used. In the end, we decided for metamodel augmentation as we need a common basis to provide general methods to store evolution information. Moreover, each artifact language can be captured as a metamodel by suitable encoding.

With our augmentation method, we enable to model the evolution of SPL artifacts using the same language concepts. This forms the basis for *co-evolution* processes for such artifacts. In this thesis, we do not provide any methods for SPL artifact co-evolution. To this end, further methods, such as model co-evolution, need to be integrated which is a possible future research direction. We argue that our metamodel augmentation forms a suitable basis for co-evolution as all types of artifacts can use the same notion of evolution. Existing methods for artifact or model co-evolution typically need to compute differences between multiple artifact versions. This has to be done for multiple different modeling languages which is highly notation specific. This step becomes obsolete using our method as we directly encode the evolution such that differences do not need to be computed.

For TFMs, we have shown that the additional expressiveness regarding evolution can be used to perform additional analyses, e.g., preventing inconsistencies or detecting anomalies. However, defining analyses for model evolution in general is a completely new field of research. Some existing research deals with model evolution consistency in general [KKT13]. We expect that the additional information stored in augmented metamodels simplifies such general analyses. Additionally, we expect that entirely new analyses, similar to evolution paradoxes for feature models, can be devised as well.

8.2.5. Guidance for Configuration Evolution

With guidance for configuration evolution, our goal is to preserve product behavior of existing configurations. We deliberately chose a very conservative notion of product behavior preservation, i.e., if artifacts remain unchanged. However, it is very unlikely that all artifacts of a product remain unchanged after evolution. Our methodology still provides the basis for the integration of more sophisticated behavior analyses. Behavior equality of different artifacts is an entirely own research domain. Respective approaches that have been devised can be integrated with our methodology as only the notion of behavior preservation has to be exchanged, and the remaining part of the methodology can be adopted as-is. Such an integration would be necessary to enable true co-evolution of feature models, implementation artifacts, and configurations.

Another aspect that we excluded is configuration validity as we abstracted from feature-model constraints. When devising configuration update operations, it can become very complex for SPL engineers to manually consider configuration validity. For large SPLs, this is even infeasible due to

the huge configuration space. Thus, additional tool support is required that can compute the impact of configuration update operations, i.e., which configurations would become invalid. Existing research mainly focuses on fixing configurations, but neglects product behavior [WSB+08, WPX+13, XHS+12, NSS16]. We argue that integrated methods addressing both topics are necessary which require extensive tool support. Together with the above-mentioned in-depth analyses of behavior preservation and augmented metamodels, this would result in true co-evolution of SPL artifacts.

Another question which comes to mind is whether our methodology provides benefit that goes beyond state-of-practice methods, i.e., changelogs that informally describe changes of a new version. In this thesis, we cannot provide an answer to that question. We know from existing work that changelogs are not sufficient for updating configurations and lead to misconfiguration of products [XZH+13]. Our methodology provides the capabilities to go beyond. Of course, we do not know whether SPL engineers use these capabilities in a more disciplined way than changelogs. As we provide a significantly more structured approach, we expect that domain engineers are encouraged to provide at least more information. Moreover, we expect that each piece of information helps product engineers to update their configurations in a better way and that our methodology provides a more user-friendly approach to update configurations. However, this all depends on how engineers use our methodology and whether our presented processes fit in their workflow. Moreover, the motivation of engineers to use our method also depends on the benefit it provides – not only in terms of better updating of configurations but also in terms of saving time. Our experience shows that saving time is one of the most motivating factors to adopt new methods. We expect that our method saves time for both engineer roles. However, this depends on the size of the SPL, how many engineers are involved, and how they can already communicate.

8.3. Possible Future Research Areas

This thesis presents methods for modeling consistent SPL evolution. This contribution forms the basis for potential further research and application areas that go beyond the scope of this thesis. In this section, we elaborate on ideas for extensions to our method with a particular focus on challenges when realizing them.

8.3.1. Collaborative TFM Development using Branching

The concept of TFMs we presented in this thesis enables to capture, plan, and replan feature-model evolution. However, in real-world development projects, it is often necessary to devise multiple development branches that each capture different (future) aspects of a feature model. To support such a development process, TFMs need to be extended to enable multiple branches including merging methods of these branches. Appropriate methods must support conflict detection and conflict resolution strategies. Similarly, multiple engineers are typically involved in TFM evolution which need to collaborate. These engineers potentially modify a TFM in parallel, which is technically similar to branches of a TFM. For true collaborative development, conflicts must not only be detected but even prevented.

Existing research provides first approaches to support collaborative development of simple feature models [KKK+19]. With TFMs another dimension of complexity is introduced as temporal validities of elements may change as well. This also gives rise to new questions and design decisions, for instance, if a feature in a TFM is moved by one engineer and another engineer post-

pones the time point of this feature's introduction to a later point in time. An appropriate solution needs to decide whether these changes are conflicting, or whether the feature is already moved to the new location when it is introduced.

Branching in TFM development additionally increases the complexity to detect evolution paradoxes. For instance, if multiple branches of a TFM exist in a development project, all branches need to be paradox-free. However, analyzing each branch on its own can become very inefficient, especially, if the feature models in the branches share a lot of data, e.g., structures or evolution operations. Thus, analyses methods that make use of this information are required. We expect that branching-time logics [EH85], such as Computation Tree Logic (CTL) [HR04], are suitable candidates to encode and analyze branched TFMs.

For branching in TFM development, it is pivotal to know whether modifications to one branch lead to evolution paradoxes in combination with another branch. If such evolution paradoxes would be detected only when actively merging two branches, many changes that have been implemented a long time ago might have to be reverted in order to provide a solution. Consequently, methods to detect evolution paradoxes in branches do not only need to consider each branch individually, but also merged branches. Optimally, such analyses are continuously performed for each change to any TFM branch. However, we expect such analyses to be computationally very complex.

Finally, in collaborative development, it is always the question who is responsible for which parts of an artifact. An important subsequent question is who is allowed to change which parts of an artifact. Consequently, access management using a model of roles is a sensible extension to TFMs but also to other models, such as the augmented metamodel we present in Chapter 6. Typically, access management is performed on file basis. However, this is not sufficient for feature models or other models. For instance, in a feature model, a certain subtree contains safety-critical features, whereas another subtree contains entertainment features. Engineers that implement entertainment features should typically not be allowed to change safety-critical features. The same applies for other models as well. Thus, capabilities to restrict model access on model element level are required. With such capabilities, it would be possible to define that certain elements, such as features of a subtree, may only be modified by engineer roles with respective rights.

8.3.2. Exploiting Evolution Information for Analyses

In our analyses to prevent the introduction of evolution paradoxes and to detect and explain anomalies, we make use of information on feature-model evolution contained in TFMs. We expect that we only scratch the surface and that a high potential exists to improve existing analyses or to define entirely new analyses. One promising idea is to only consider changes of a TFM that may contribute to an evolution paradox or anomaly. For instance, if an intermediate evolution operation is devised, the analyses to prevent evolution paradox introduction only need to check those operations that potentially conflict with the new intermediate operation. For anomalies, changes result in new or removed clauses for a respective solver. New anomalies might be detected by analyzing which clauses of future points in time are related to the clauses of the new change, i.e., if they (transitively) contain the same literals. As a result, analyses incorporating more evolution information of a TFM bear the potential for a significant increase in efficiency.

Entirely new analyses might enable engineers to maintain TFMs more efficiently. For instance, for anomalies, we currently generate explanations in terms of causing evolution operations at the

time point of anomaly introduction. However, it may be the case that fixing an anomaly using such an explanation at the time point of anomaly introduction results in the anomaly to be still existent at future time points. Such a case occurs if different causes for an anomaly at different points in time exist. A resulting question is what is the identity of an anomaly. One could argue that this is the same anomaly as that feature is dead for subsequent points in time, or one could argue that these are different anomalies as they have different explanations. If the latter is assumed, analyses are required that compute stable anomaly explanations for multiple evolution steps to define fixes that resolve the anomaly for all time points.

8.3.3. Repairing Evolution Paradoxes

We guarantee TFMs to be free from evolution paradoxes by preventing the introduction of paradox-causing evolution operations. This is a very strict method and might limit engineers in designing a TFM in the desired way. To overcome this strict limitation, methods that temporarily allow the introduction of evolution paradoxes, but enable to repair them subsequently are required. To this end, multiple contributions are required. First, engineers need to understand why an evolution operation would introduce an evolution paradox. Consequently, methods are necessary that explain evolution paradoxes – similar to explanations for anomalies. This forms the basis for engineers to provide suitable fixes.

Second, engineers need to be supported with strategies to fix inconsistencies. Multiple strategies are conceivable. First, the intermediate, paradox-causing operation can be undone. This is the same as our current approach, but is very limiting. Second, the future evolution operation which conflicts with the intermediate operation can be undone. However, this may entail multiple changes as other evolution operations might be based on the changes that are introduced by the conflicting operation. Third, evolution paradoxes can be temporarily ignored but a constraint on the TFM evolution must be defined stating that before the conflicting operation becomes active, the conflicts must be resolved. Thus, engineers have time to fix that inconsistency until the conflicting operation becomes active. As an extension to the third possibility, guidance in terms of automatically computed solution paths can be provided. Creating methods to compute such paths is extremely complex as hypothetical operations must be computed and it must be evaluated whether these operations repair the evolution paradox. However, as the possibilities to change a TFM are theoretically unlimited, smart heuristics need to limit the search space and find suitable solution in a sensible amount of time.

8.3.4. Co-Evolution of SPL Artifacts

A holistic co-evolution of SPL artifacts is a key challenge in SPL engineering. With our methods to augment arbitrary models, we provide a starting point in terms of the same syntax for such a co-evolution. However, many more aspects need to be considered. In particular, the impact of changes of one artifact, such as the TFM, on other artifacts, such as realization artifacts or configurations. For instance, feature interaction occurs if the presence of two features in a product at the same time results in different behavior than their individual presence. The TFM may prohibit the simultaneous selection of two features using constraints. However, after TFM evolution, their combined selection may be allowed. As a result, the realization artifacts for those features need to be capable of dealing with the presence of the respective other. The same applies also the other way

round, e.g., if code responsible for feature interaction is changed or removed, constraints to prohibit the combined selection of the affected features may have to be integrated in the TFM. Consequently, changes to one artifact have to be traced other to linked artifacts, and respective information must be presented to engineers. Industry already acknowledged the necessity of such analyses and the benefit of respective information [Liv11].

Part V.

Appendix

A Transformation Rules for Meta-model Augmentation

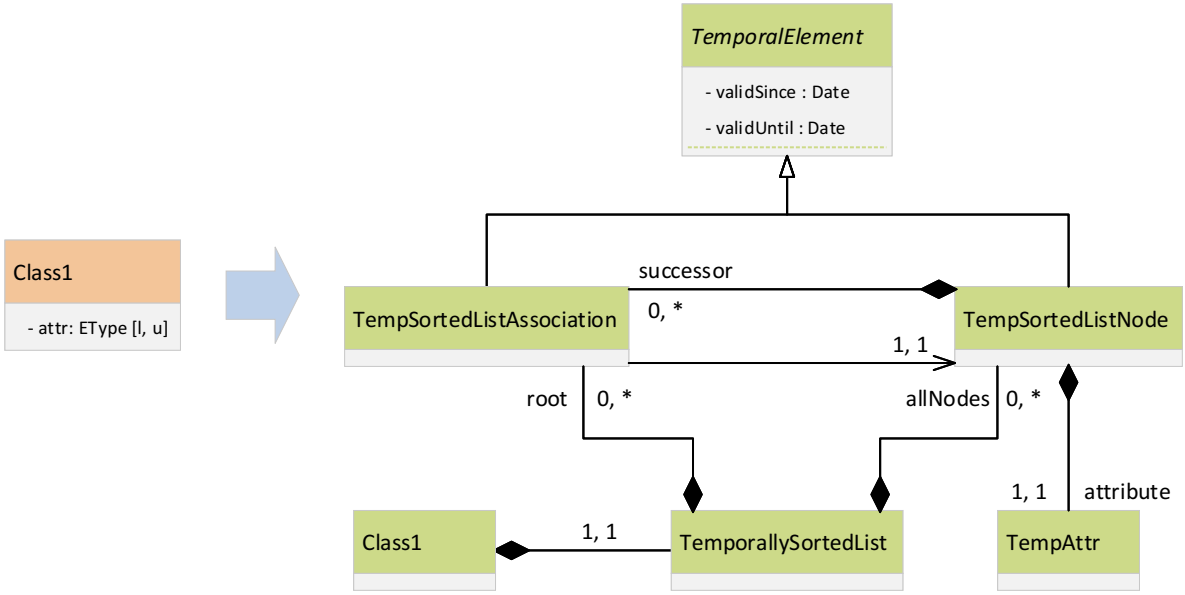


Figure A.1.: Transformation Rule for Augmenting Metamodels with Sorted Attribute List Evolution.

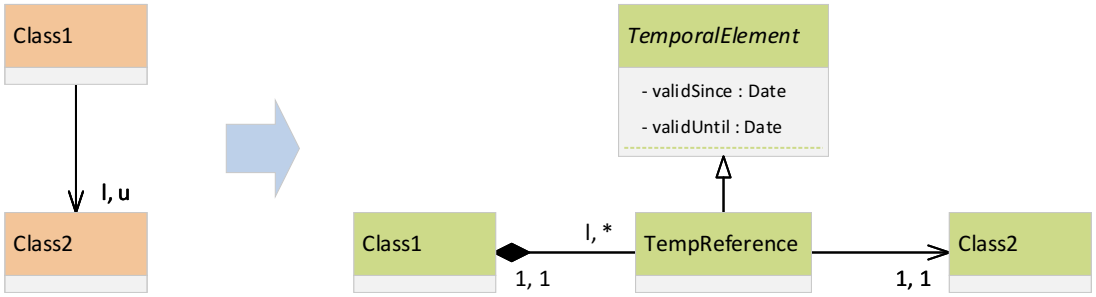


Figure A.2.: Transformation Rule for Augmenting Metamodels with Unidirectional Reference Evolution.

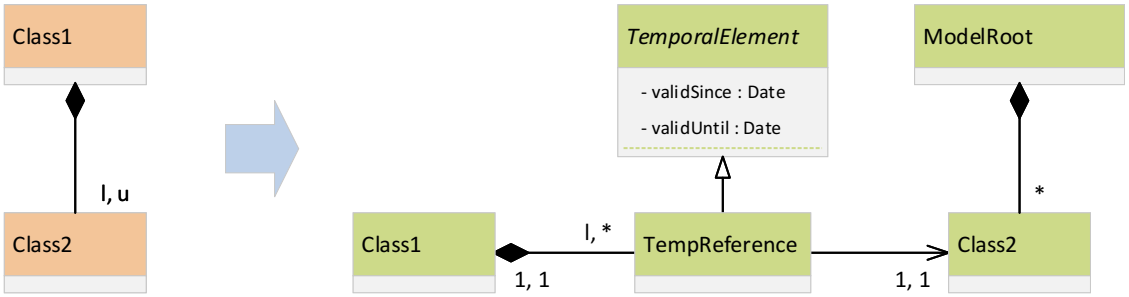


Figure A.3.: Transformation Rule for Augmenting Metamodels with Unidirectional Containment Reference Evolution.

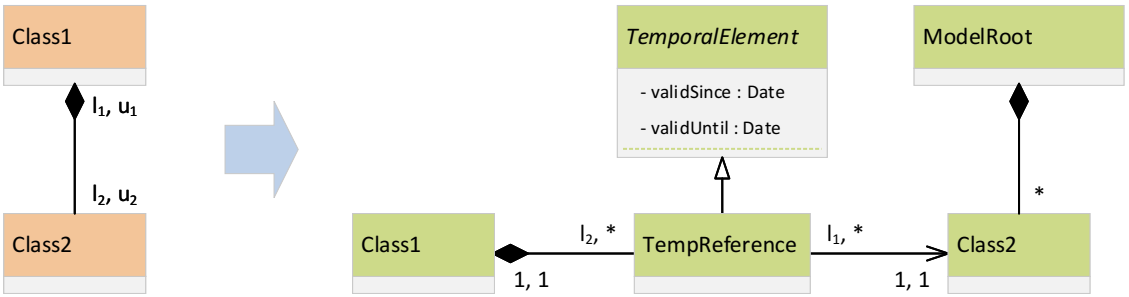


Figure A.4.: Transformation Rule for Augmenting Metamodels with Bidirectional Containment Reference Evolution.

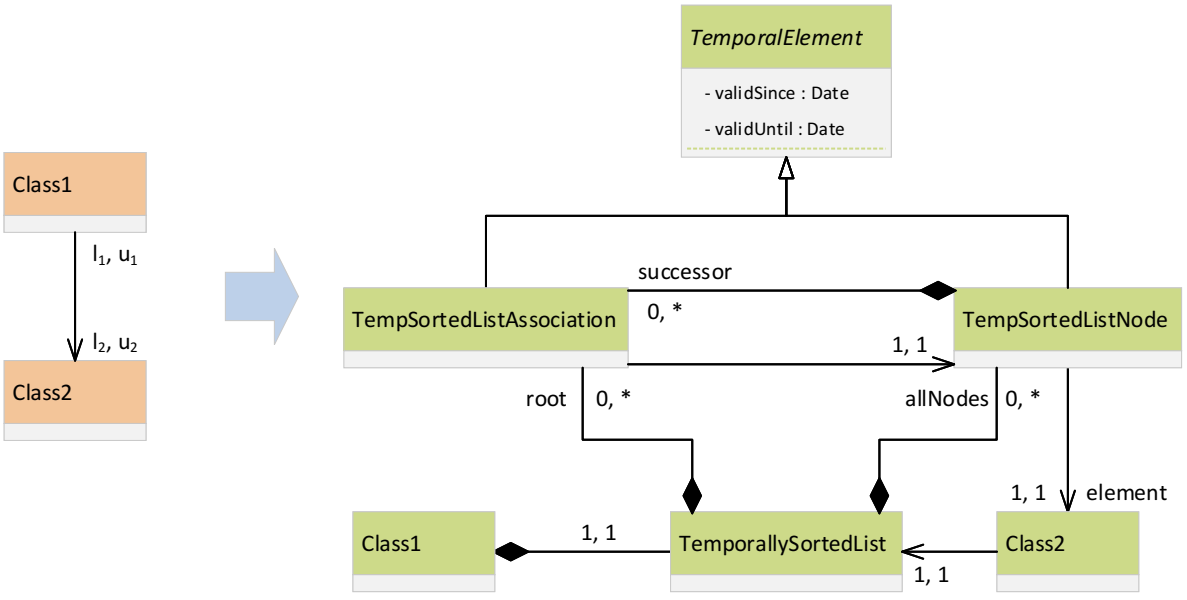


Figure A.5.: Transformation Rule for Augmenting Metamodels with Sorted Bidirectional Reference Evolution.

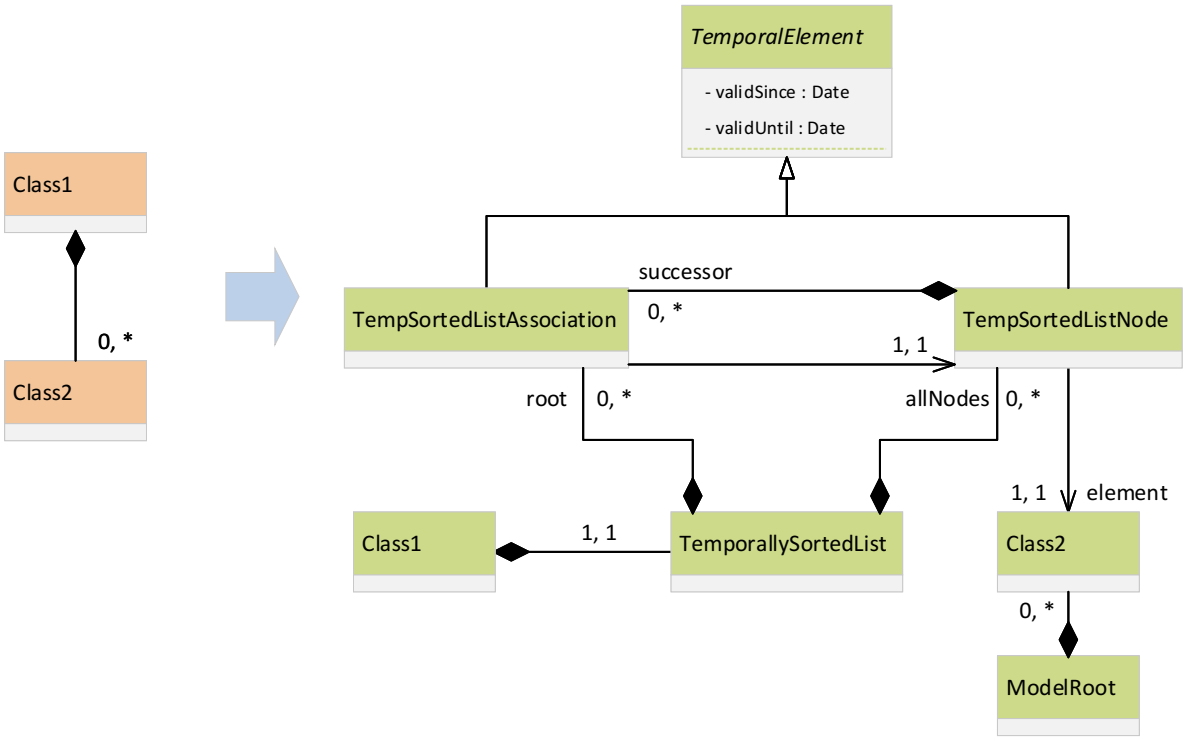


Figure A.6.: Transformation Rule for Augmenting Metamodels with Sorted Unidirectional Containment Reference Evolution.

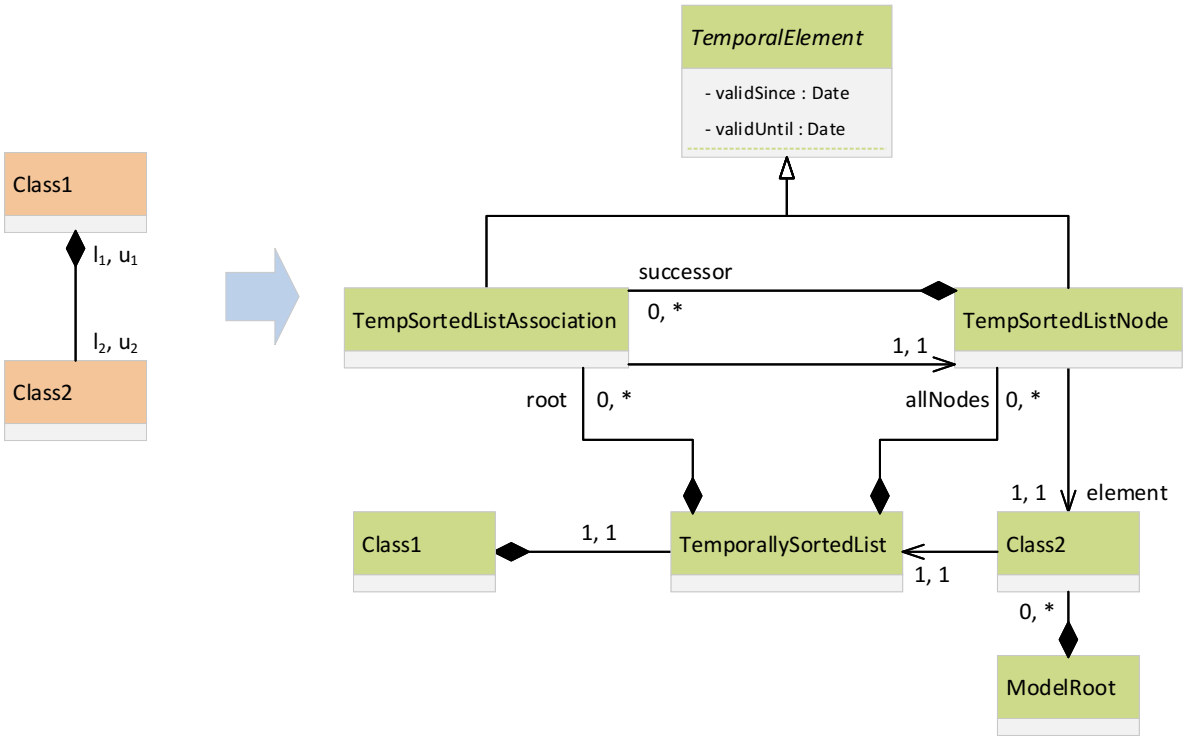


Figure A.7.: Transformation Rule for Augmenting Metamodels with Sorted Bidirectional Containment Reference Evolution.

B Paradox-Causing Evolution Operations and Semantics Rules for Evolution Operations

B.1. Paradox-Causing Evolution Operations

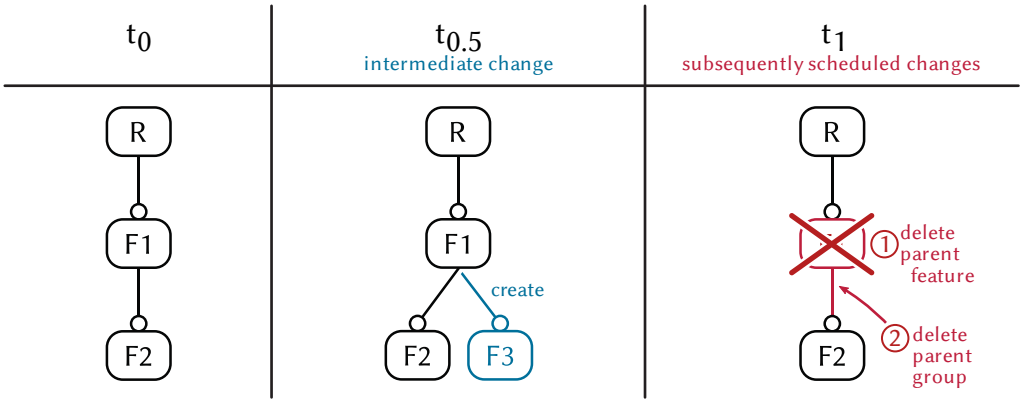


Figure B.1.: Graphical representation under which circumstances an intermediate *create feature* evolution operation scheduled for $t_{0.5}$ causes an evolution paradox at t_1.

Figure B.1 visualizes in which situations an intermediate *feature create* operation causes an evolution paradox. The arrows ①, and ② represent evolution operations scheduled for a subsequent point in time. These operations cancel the effect of the *create feature*. Consequently, in both cases, a *Transient Effect Paradox* occurs.

Figure B.2 visualizes in which situations an intermediate *feature move* operation causes an evolution paradox. The arrows ①, ②, and ③ represent evolution operations scheduled for a subsequent point in time. These operations conflict with or cancel the effect of the retroactively introduced intermediate *delete feature* operation:

① A *Variation Type Paradox* occurs if a type change operation to OR or ALTERNATIVE of the moved feature's old parent group has been defined previously but scheduled for a subsequent point in time and that group only contains two child features.

② A *Variation Type Paradox* occurs if the moved feature's type is mandatory (or is changed at a subsequent point in time to mandatory) and the moved feature's target group's type is changed to OR or ALTERNATIVE.

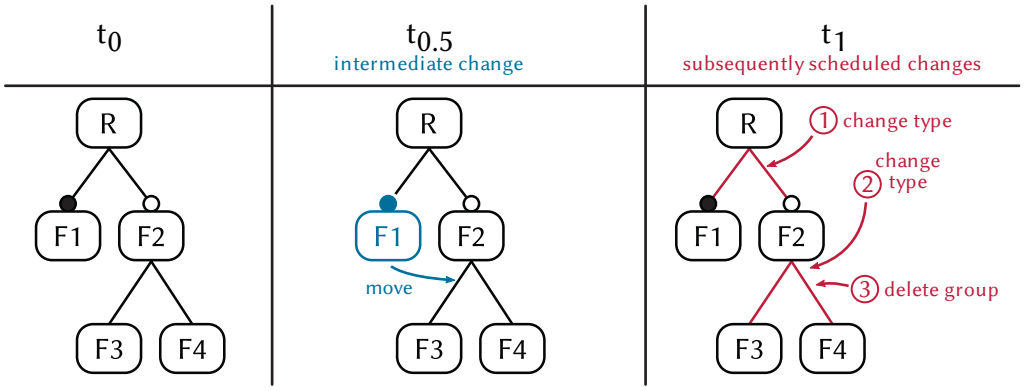


Figure B.2.: Graphical representation under which circumstances an intermediate *move feature* evolution operation scheduled for $t_{0.5}$ causes an evolution paradox at t_1.

③ A *Transient Effect Paradox* occurs if the moved feature's target group is deleted at a subsequent point in time.

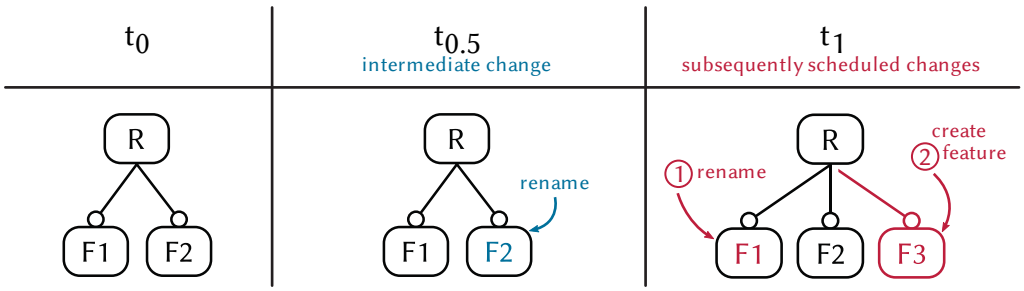


Figure B.3.: Graphical representation under which circumstances an intermediate *rename feature* evolution operation scheduled for $t_{0.5}$ causes an evolution paradox at t_1.

Figure B.3 visualizes in which situations an intermediate *feature rename* operation causes an evolution paradox. The arrows ①, and ② represent evolution operations scheduled for a subsequent point in time. These operations conflict with the retroactively introduced intermediate *rename feature* operation and introduce a *Naming Conflict Paradox*, if another feature (`F1` in the diagram) is renamed (①) to the same name, or if a feature (`F3`) is created (③).

Figure B.4 visualizes in which situations an intermediate *feature type change* operation causes an evolution paradox. The arrows ①, and ② represent evolution operations scheduled for a subsequent point in time. These operations conflict with the retroactively introduced intermediate *feature type change* operation and introduce a *Variation Type Paradox*, if: ① the feature's parent group's type is changed to Or or alternative at a subsequent point in time, or ② the type is changed to mandatory in the intermediate operation and the feature is moved to an or or alternative group at a subsequent point in time.

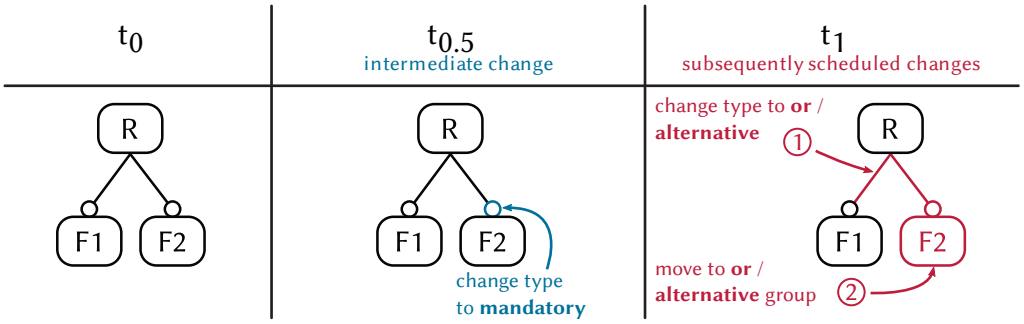


Figure B.4.: Graphical representation under which circumstances an intermediate *feature type change* to MANDATORY evolution operation scheduled for $t_{0.5}$ causes an evolution paradox at t_1.

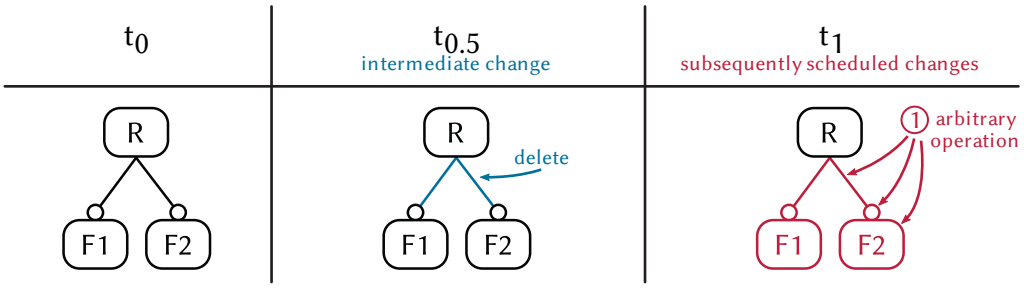


Figure B.5.: Graphical representation under which circumstances an intermediate *group delete* evolution operation scheduled for $t_{0.5}$ causes an evolution paradox at t_1.

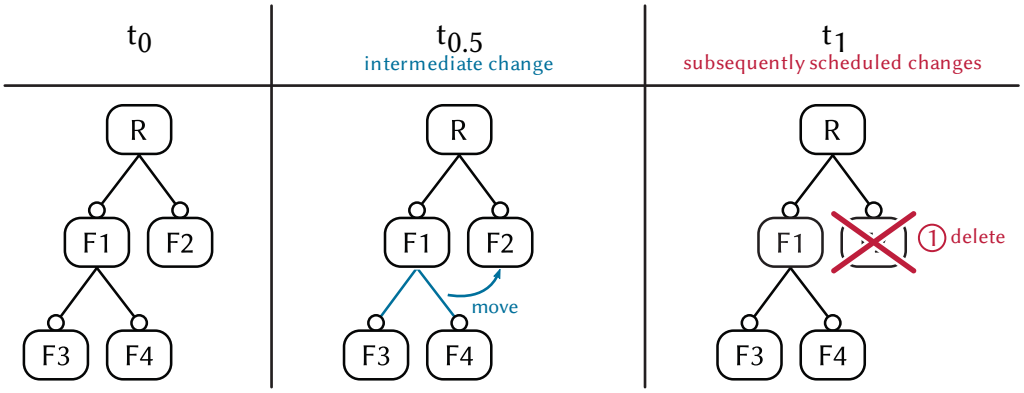


Figure B.6.: Graphical representation under which circumstances an intermediate *group move* evolution operation scheduled for $t_{0.5}$ causes an evolution paradox at t_1.

Figure B.5 visualizes in which situations an intermediate *group delete* operation causes an evolution paradox. The arrow ① represents evolution operations scheduled for a subsequent point in time. These operations conflict with the retroactively introduced intermediate *group delete* operation and introduce a *Non-Existent Element Edit Paradox*, if: the group is modified in any way or if a feature of the group's sub features (in the entire sub tree) is modified in any way.

Figure B.6 visualizes in which situations an intermediate *group move* operation causes an evolution paradox. The arrow ① represents evolution operations scheduled for a subsequent point in time. These operations cancel the effect of the retroactively introduced intermediate *group*

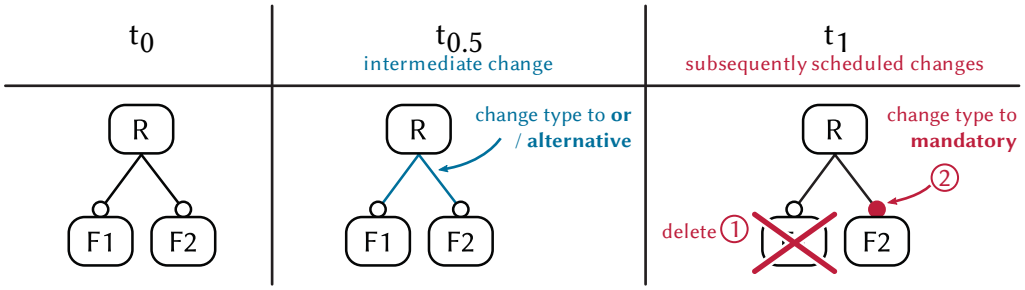


Figure B.7.: Graphical representation under which circumstances an intermediate *group type change* to OR or ALTERNATIVE evolution operation scheduled for $t_{0.5}$ causes an evolution paradox at t_1.

move operation and, thus, introduce a *Transient Effect Paradox* if the moved group's target feature is deleted in a subsequent evolution operation.

Figure B.7 visualizes in which situations an intermediate *group type change* operation to OR or ALTERNATIVE causes an evolution paradox. The arrows ①, and ② represent evolution operations scheduled for a subsequent point in time. These operations conflict with the retroactively introduced intermediate *group type change* operation and introduce a *Variation Type Paradox*, if: ① at a subsequent point in time, the group contains only two features from which one is deleted, or ② the type of a group feature is changed to MANDATORY at a subsequent point in time.

B.2. Semantics Rules for Evolution Operations

The feature delete operation (Semantics Rule B.1) can only be executed if and only if the preconditions above the line hold, i.e.:

- the feature to be deleted is not the root feature,
- the feature to be deleted exists in the feature table,
- the feature to be deleted does not have subgroups

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature is removed as new child feature of the specified group and a new feature table entry $[FeatureID \mapsto (Name, ParentFeatureID, \emptyset, FType)]$ is added to FT with an empty set for the child group IDs.

Semantics Rule B.1 (Feature Delete Operation)

$$\frac{\begin{array}{c} RemoveFid \neq RootId \\ FT = FT' + [RemoveFid \mapsto (Name, ParentFid, \emptyset, FType)] \\ FT'' = removeFeatureFromParent(FT', ParentFid, RemoveFid) \end{array}}{\begin{array}{c} FM(RootId, FT) \\ \textbf{\textit{deleteFeature}}(RemoveFid) \\ \Rightarrow \\ FM(RootId, FT'') \end{array}}$$

The feature move operation (Semantics Rule B.2) can only be executed if and only if the preconditions above the line hold, i.e.:

- the feature to be moved is not the root feature,

- the new parent feature is no subfeature of the moved feature,
- the feature to be moved exists in the feature table,
- the feature's type is valid after it has been moved (i.e., Optional if moved into Alternative or Or group)

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature is removed from its old parents (FT'') and is then added to the new target group *NewGroup* (FT''').

Semantics Rule B.2 (Feature Move Operation)

$$\frac{\begin{array}{c} MoveFid \neq RootId \\ NewParent = parentOfGroup(FT, NewGroup) \\ \neg isSubFeature(MoveFid, NewParent, RootId, FT) \\ FT = FT' + [MoveFid \mapsto (Name, ParentFid, Groups, FType)] \\ FT'' = removeFeatureFromParent(FT', ParentFid, MoveFid) \\ FT''' = createFeatureToGroup(FT'', NewGroup, MoveFid) \\ isValidType(FT''', MoveFid) \end{array}}{\begin{array}{c} FM(RootId, FT) \\ \textbf{\textit{moveFeature}}(MoveFid, NewGroup) \\ \Rightarrow \\ FM(RootId, FT''' + [MoveFid \mapsto (Name, NewParent, Groups, FType)]) \end{array}}$$

The feature rename operation (Semantics Rule B.3) can only be executed if and only if the pre-conditions above the line hold, i.e.:

- the new name is unique in the feature model,
- the feature exists in the feature table

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature table entry with the old feature name is removed (FT') and a new entry with the new feature name is added (FT'').

Semantics Rule B.3 (Feature Rename Operation)

$$\frac{\begin{array}{c} isUniqueName(NewName, FT) \\ FT = FT' + [TargetFid \mapsto (Name, Parent, Groups, FType)] \\ FT'' = FT' + [TargetFid \mapsto (NewName, Parent, Groups, FType)] \end{array}}{\begin{array}{c} FM(RootId, FT) \\ \textbf{\textit{renameFeature}}(TargetFid, NewName) \\ \Rightarrow \\ FM(RootId, FT'') \end{array}}$$

The feature type change operation (Semantics Rule B.4) can only be executed if and only if the pre-conditions above the line hold, i.e.:

- the considered feature is not the root feature,
- the feature exists in the feature table,
- the feature type is valid (i.e., Optional if in an Alternative or Or group)

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature table entry of feature *TargetFeature* is removed (FT') and a new entry with the new feature type is added (FT'').

Semantics Rule B.4 (Feature Type Change Operation)

$$\frac{\begin{array}{c} TargetFid \neq RootId \\ FT = FT' + [TargetFid \mapsto (Name, Parent, Groups, Type)] \\ FT'' = FT' + [TargetFid \mapsto (Name, Parent, Groups, NewType)] \\ isValidType(FT'', TargetFid) \end{array}}{\begin{array}{c} FM(RootId, FT) \\ \textbf{\textit{changeFeatureType}}(TargetFid, NewType) \\ \Rightarrow \\ FM(RootId, FT'') \end{array}}$$

The group create operation (Semantics Rule B.5) can only be executed if and only if the preconditions above the line hold, i.e.:

- the group ID is unique,
- the parent feature ID (*TargetFid*) exists in the feature table

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature table entry of the parent feature is updated such that the new group is added to its subgroups (FT'').

Semantics Rule B.5 (Group Create Operation)

$$\frac{\begin{array}{c} isUniqueGroupId(GroupId, FT) \\ FT = FT' + [TargetFid \mapsto (Name, Parent, Groups, FType)] \\ FT'' = FT' + [TargetFid \mapsto (Name, Parent, \{(GroupId, GType, \emptyset)\} \cup Groups, FType)] \end{array}}{\begin{array}{c} FM(RootId, FT) \\ \textbf{\textit{createGroup}}(TargetFid, GroupId, GType) \\ \Rightarrow \\ FM(RootId, FT'') \end{array}}$$

The group delete operation (Semantics Rule B.6) can only be executed if and only if the preconditions above the line hold, i.e.:

- the group exists in the feature table,
- the group's child feature set is empty

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature table entry of the group's parent feature is updated such that the group is removed from the set of the parent feature's subgroups (FT'').

Semantics Rule B.6 (Group Delete Operation)

$$\frac{\begin{array}{c} FT = FT' + [TargetFid \mapsto (Name, Parent, \{(GroupId, GType, \emptyset)\} \cup Groups, FType)] \\ FT'' = FT' + [TargetFid \mapsto (Name, Parent, Groups, FType)] \end{array}}{\begin{array}{c} FM(RootId, FT) \\ \textbf{\textit{deleteGroup}}(GroupId) \\ \Rightarrow \\ FM(RootId, FT'') \end{array}}$$

The group type change operation (Semantics Rule B.7) can only be executed if and only if the pre-conditions above the line hold, i.e.:

- the group exists in the feature table,
- the types of all of the group's subfeatures are valid with the new group's type (i.e., Optional if new group type is Alternative or Or group)

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature table entry of the group's parent feature is updated such that the group is removed from the set of the parent feature's subgroups and the same group but with the modified type is added to the subgroup set (FT'').

Semantics Rule B.7 (Group Type Change Operation)

$$\frac{\begin{array}{c} FT = FT' + [ParentFid \mapsto (Name, Parent, \{(GroupId, Type, Features)\} \cup Groups, FType)] \\ FT'' = FT' + [ParentFid \mapsto (Name, Parent, \{(GroupId, NewType, Features)\} \cup Groups, FType)] \\ \forall feature \in Features \cdot isValidType(FT'', feature) \end{array}}{\begin{array}{c} FM(RootId, FT) \\ \textbf{\textit{changeGroupType}}(GroupId, NewType) \\ \Rightarrow \\ FM(RootId, FT'') \end{array}}$$

The group move operation (Semantics Rule B.8) can only be executed if and only if the pre-conditions above the line hold, i.e.:

- the target feature is not in the subtree of the considered group,
- the group exists in the feature table

If all preconditions are met, the effect of the operation can be applied to the feature table FT (below the line): The feature table entry of the group's parent feature is updated such that the group is removed from the set of the old parent feature's subgroups and the same group is added to the subgroup set of the target feature (FT''). Additionally, all parent feature references of the group's direct subfeatures are update to the group's new parent feature.

Semantics Rule B.8 (Group Move Operation)

$$\frac{\begin{array}{c}\forall feature \in Features \cdot \neg isSubFeature(feature, newParentFid, RootId, FT) \\ FT = FT' + [OldParentFid \mapsto (Name', Parent', \{(GroupID, GType, Features)\} \cup Groups', FType')] \\ +[NewParentFid \mapsto (Name, Parent, Groups, FType)] \\ FT'' = FT' + [OldParentFid \mapsto (Name', Parent', Groups', FType')] \\ +[NewParentFid \mapsto (Name, Parent, \{(GroupID, GType, Features)\} \cup Groups, FType)] \\ FT''' = updateParents(FT'', Features, NewParentFid)\end{array}}{\begin{array}{c}FM(RootId, FT) \\ \textbf{\textit{moveGroup}}(GroupID, NewParentFid) \\ \Rightarrow \\ FM(RootId, FT''')\end{array}}$$

C Templates for the Empirical Evaluation of the Paradox-Free Feature-Model Evolution Planning

C.1. Guide for Semi-Structured Interview with Industry Experts

Translated from German.

Questions
Welcoming and explanation of this interview
What are your touch points with feature-oriented development? How do you use it in your business?
Present introductory slides on SPLs, FMs and their evolution.
Have you ever been involved in planning an SPL? (How did you plan it? What kind of SPL was it? What size was it?)
Software and, thus, SPLs evolve in a continuous fashion. What are driving factors for software evolution in your company and how does this evolution take place (where does it start, how does it propagate)?
How do you plan SPL evolution?
How often do you change an SPL development plan?
How do you monitor the compliance with an SPL development plan?
How often do you change an FM?

Present slides on our concept of FM evolution plans for SPL planning.
Do you use FMs for planning purposes? (If so, do you plan multiple versions ahead?)
How suitable do you assess an FM evolution plan for SPL planning?
Present slides on evolution paradoxes (structural inconsistencies) in FM evolution plans.
Have you ever encountered such a situation?
How important is the detection of such inconsistencies?
How valuable is a mechanism that automatically prevents the introduction of such inconsistencies?
Time for open topics, interesting points and goodbye.

C.2. Online Survey for Experts from Academia

Section 1 of 4

Feature Model Evolution Planning

Welcome to this questionnaire about evolution planning for feature-oriented development.

Topic

The idea of our approach is to plan feature-oriented development by simply modelling future states of a system in form of feature models (= feature model evolution plan).
These future versions of a feature model (FM) do not need to be implemented in terms of realization artifacts, yet: they serve as abstract plan.
Throughout the implementation of an FM evolution plan, it can be refined with details at will.

Total number of questions: 9

We have a total number of 9 questions for you, relating to that topic and challenges related to it.

Number of sections of this questionnaire: 3

These questions are devided in 3 sections. For each section, we provide scenarios to underpin our questions.

Estimated length: 10 ~ 15 minutes

For each section, it requires a little time to get familiar with the provided scenario.

General Remarks

Each section is concluded by an optional text field to give additional feedback and general remarks, concerning the questions asked in the section.

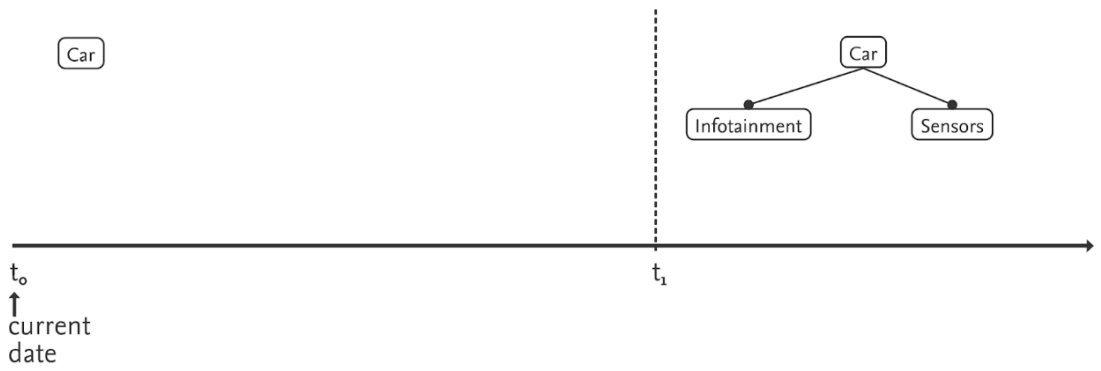
Section 2 of 4

(1) Continuous Feature Model Evolution Planning

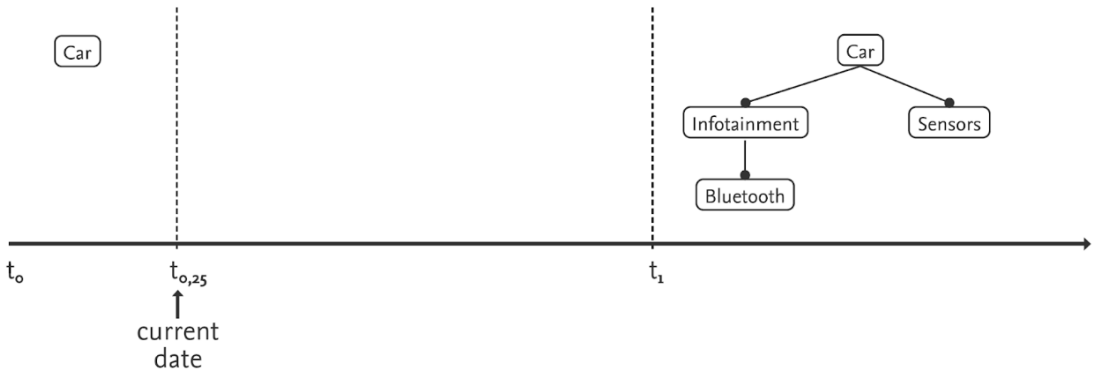
The general idea explained

Example Car System

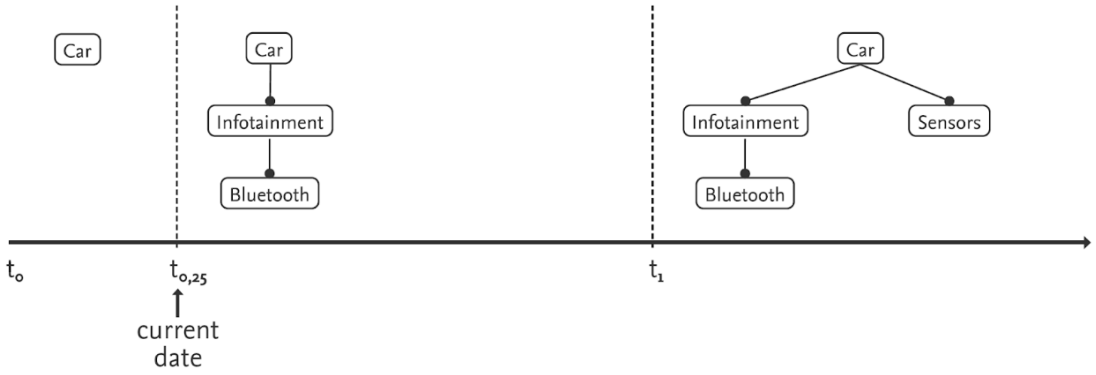
In its initial draft, the FM evolution plan for the example car system is constructed in a very coarse-grain manner: By time point t_1 (right hand side), the car system is supposed to be configurable with two features "Infotainment" and "Sensors".



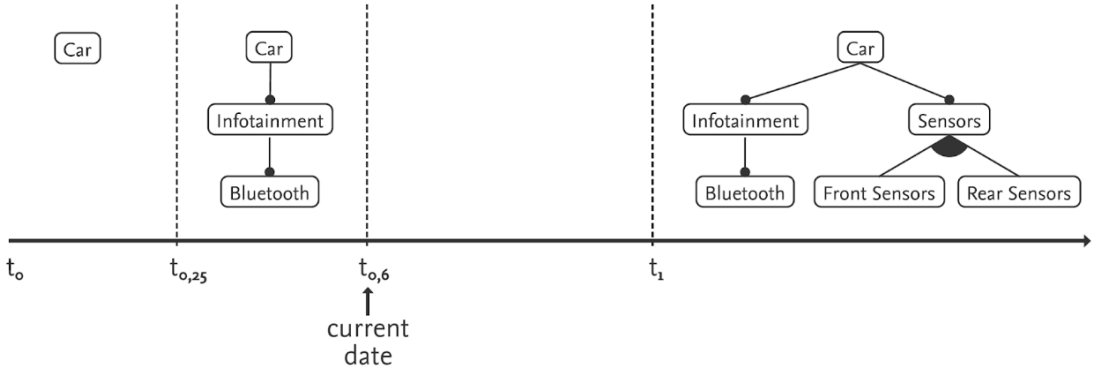
As time progresses, the planned state for t_1 is refined with more details (a new feature is added to the planned state).



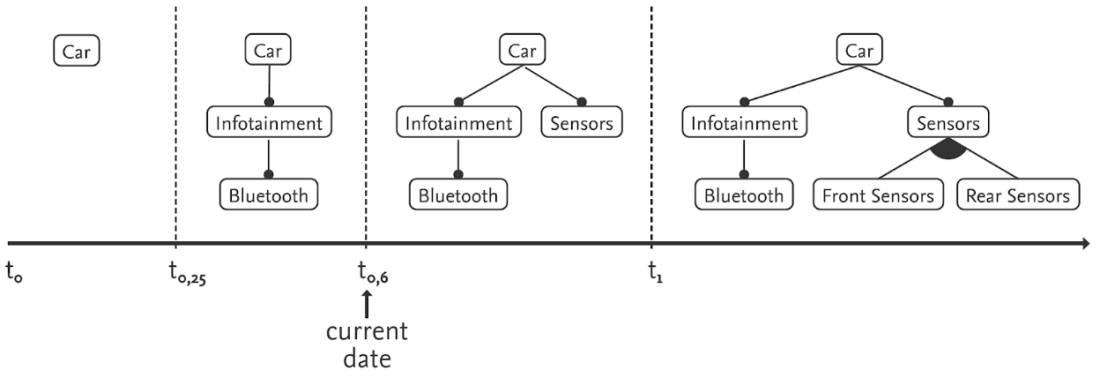
At the same time, a first increment of development work is finished, yielding first implementation artifacts. Respectively, the current FM state is adjusted to match that state.



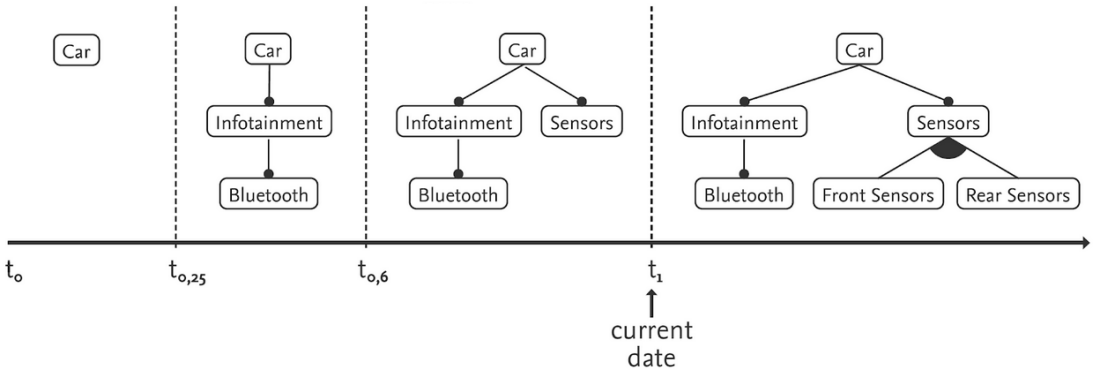
This process is repeated: with more insights into the system and its development, the future FM state gets refined...



... before extending the current state with implented functionality.



Eventually, t_1 is reached. By now, all realization artifacts for the planned feature should be implemented.



How suitable do you assess this approach of future evolution planning for ...

Description (optional)

... large teams with many developers who modify parts of the FM in parallel? *

	0	1	2	3	4	5	
not suitable	○	○	○	○	○	○	very suitable

... smaller teams with few to one stakeholder(s) involved? *

	0	1	2	3	4	5	
not suitable	○	○	○	○	○	○	very suitable

Do you have general remarks relating to this approach of evolution planning?

Long answer text

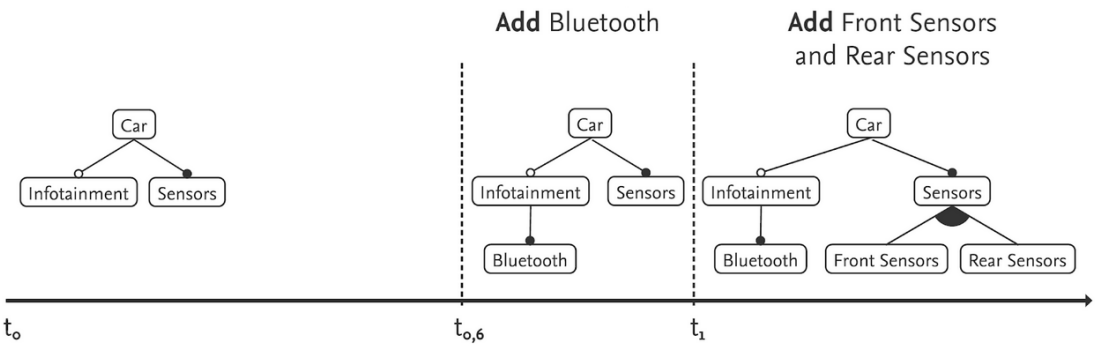
Section 3 of 4

(2) Evolution Paradoxes

An evolution plan can be replanned by changing it in an intermediate state (altering an intermediate FM version). However, this can lead to inconsistencies.

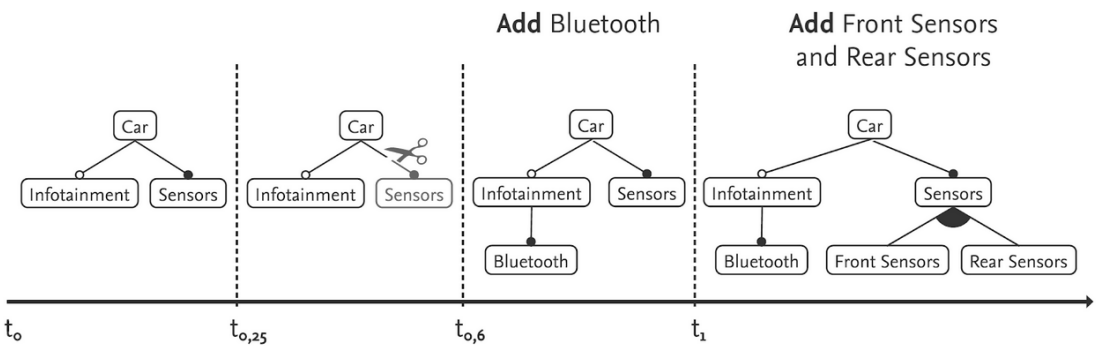
Example

Assume the following FM evolution plan:

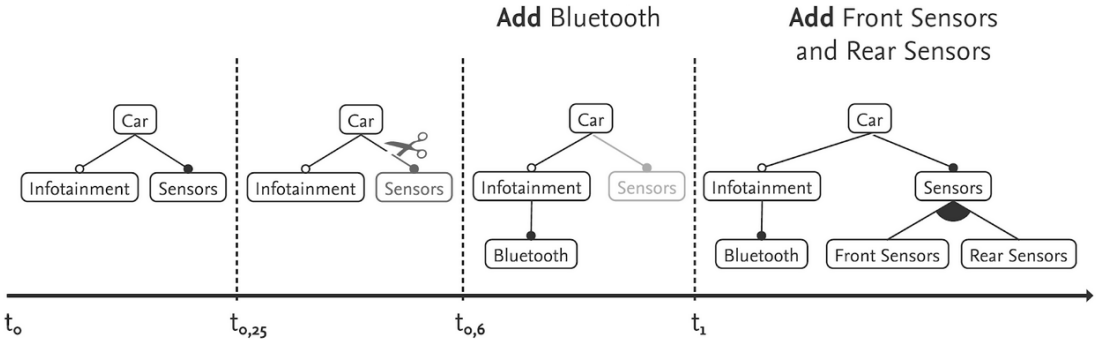


Replanning

In order to save costs, the feature "Sensors" is replanned to be removed in an intermediate evolution step (red color).

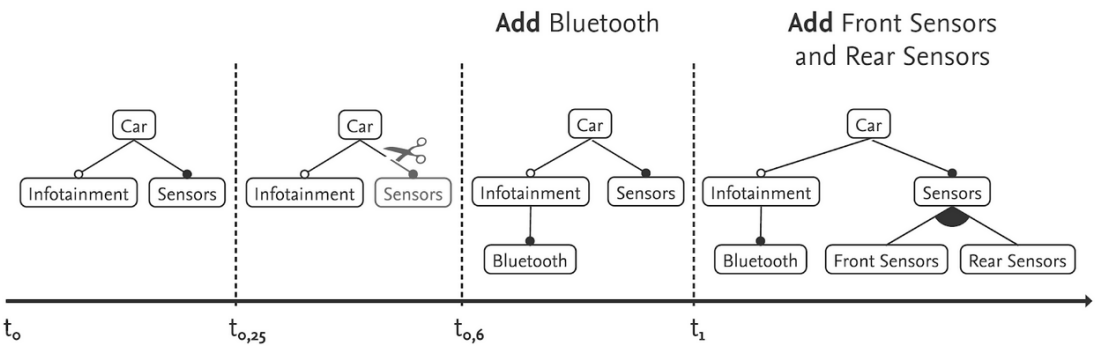


This change must now be adopted to all subsequently scheduled steps of the plan.
While this does not cause problems for version t_0,6 ...



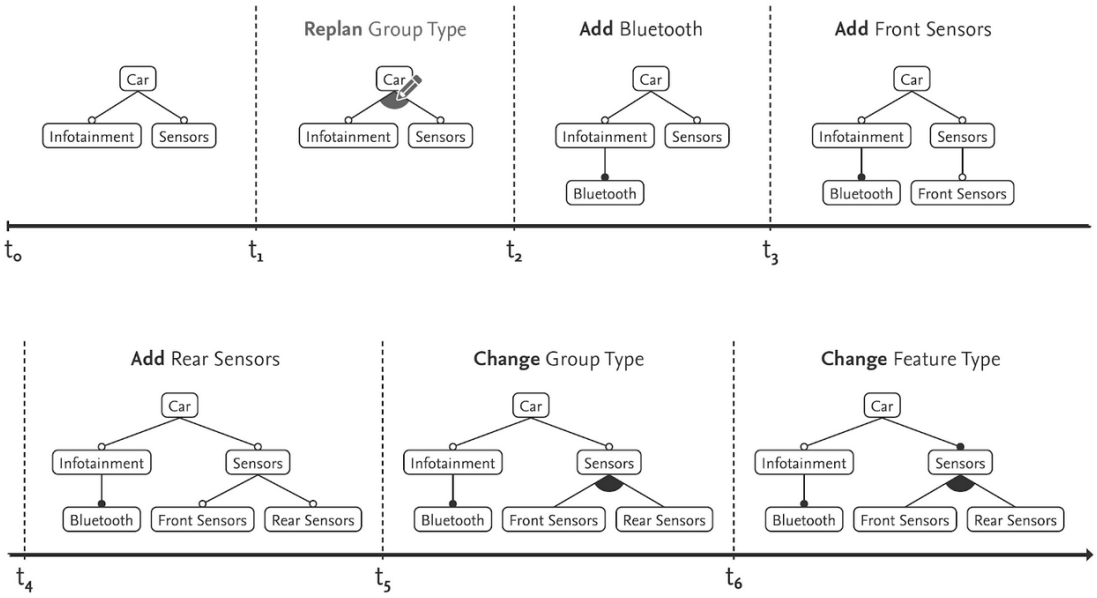
... the change cannot be adopted to t_1 as other planned changes (red color) conflict with a deletion of "Sensors".

How difficult do you assess a manual detection of the evolution paradox presented in the above example? *



	0	1	2	3	4	5	
quite easy	○	○	○	○	○	○	very difficult

How difficult do you assess a manual detection of potential evolution paradoxes in this alternative scenario? *



	0	1	2	3	4	5	
quite easy	○	○	○	○	○	○	very difficult

Do you think a manual detection of evolution paradoxes is feasible for FM evolution plans with large-scale feature models and many evolutionary changes? *

- ○ Yes, such inconsistencies are easily detectable.
- ○ No, a manual detection of such inconsistencies is too cumbersome for large models and/or numbers of ve…

Assume that you plan a variable system by means of an FM evolution plan. How do you assess the value of an automated detection mechanism for evolution paradoxes in FM evolution plans? *

	0	1	2	3	4	5	
irrelevant	○	○	○	○	○	○	crucial

Do you have general remarks concerning evolution paradoxes?

Long answer text

Bibliography

[ABK+16] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2016.

[AEH+99] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. "Graph transformation for specification and programming". In: *Science of Computer Programming* 34.1 (1999), pp. 1–54. DOI: https://doi.org/10.1016/S0167-6423(98)00023-9.

[AFV01] L. Aceto, W. Fokkink, and C. Verhoef. "Structural operational semantics". In: *Handbook of process algebra*. Elsevier, 2001, pp. 197–292.

[AGM+06] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. "Refactoring Product Lines". In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE '06. Portland, Oregon, USA: ACM, 2006, pp. 201–210. DOI: 10.1145/1173706.1173737.

[AGV17] P. Arcaini, A. Gargantini, and P. Vavassori. "Automated Repairing of Variability Models". In: *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*. SPLC '17. Sevilla, Spain: Association for Computing Machinery, 2017, pp. 9–18. DOI: 10.1145/3106195.3106206.

[AKT+16] S. Ananieva, M. Kowal, T. Thüm, and I. Schaefer. "Implicit Constraints in Partial Feature Models". In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. FOSD 2016. Amsterdam, Netherlands: ACM, 2016, pp. 18–27.

[Bat04] D. Batory. "Feature-oriented programming and the AHEAD tool suite". In: *Proceedings. 26th International Conference on Software Engineering*. 2004, pp. 702–703.

[Bat05] D. Batory. "Feature Models, Grammars, and Propositional Formulas". In: *Proceedings of the 9th International Conference on Software Product Lines*. SPLC'05. Rennes, France: Springer-Verlag, 2005, pp. 7–20. DOI: 10.1007/11554844_3.

[BCM+04] G. Bockle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid. "Calculating ROI for software product lines". In: *IEEE Software* 21.3 (May 2004), pp. 23–31. DOI: 10.1109/MS.2004.1293069.

[BHM09] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[BKL+12] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. "An Introduction to Model Versioning". In: *Proceedings of the 12th Intl. Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*. Bertinoro, Italy: Springer-Verlag, 2012, pp. 336–398.

[BKL+16] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr. "Reasoning about product-line evolution using complex feature model differences". In: *Automated Software Engineering* 23.4 (2016), pp. 687–733.

[BNR+14] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. "Three Cases of Feature-Based Variability Modeling in Industry". In: *Model-Driven Engineering Languages and Systems*. Ed. by J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran. Cham: Springer International Publishing, 2014, pp. 302–319.

[Boe84] B. W. Boehm. "Software Engineering Economics". In: *IEEE Transactions on Software Engineering* SE-10.1 (Jan. 1984), pp. 4–21. DOI: 10.1109/TSE.1984.5010193.

[Bos01] J. Bosch. "Software product lines: organizational alternatives". In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. May 2001, pp. 91–100. DOI: 10.1109/ICSE.2001.919084.

[BP08] C. Brun and A. Pierantonio. "Model differences in the eclipse modeling framework". In: *UPGRADE, The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34.

[BP14] G. Botterweck and A. Pleuss. "Evolution of Software Product Lines". In: *Evolving Software Systems*. Ed. by T. Mens, A. Serebrenik, and A. Cleve. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 265–295. DOI: 10.1007/978-3-642-45398-4_9.

[BPD+10] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski. "EvoFM: feature-driven planning of product-line evolution". In: *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*. ACM. 2010, pp. 24–31.

[BPP+09] G. Botterweck, A. Pleuss, A. Polzer, and S. Kowalewski. "Towards Feature-Driven Planning of Product-Line Evolution". In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. FOSD '09. Denver, Colorado, USA: Association for Computing Machinery, 2009, pp. 109–116. DOI: 10.1145/1629716.1629737.

[BRN+13] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. "A Survey of Variability Modeling in Industrial Practice". In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '13. Pisa, Italy: Association for Computing Machinery, 2013. DOI: 10.1145/2430502.2430513.

[BSR10] D. Benavides, S. Segura, and A. Ruiz-Cortés. "Automated analysis of feature models 20 years later: A literature review". In: *Information Systems* 35.6 (2010), pp. 615–636.

[BT18] C. Barrett and C. Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. Cham: Springer International Publishing, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.

[BTG12] P. Borba, L. Teixeira, and R. Gheyi. "A theory of software product line refinement". In: *Theoretical Computer Science* 455 (2012), pp. 2–30.

[BTR05] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. "Automated Reasoning on Feature Models". In: *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*. CAiSE'05. Porto, Portugal: Springer-Verlag, 2005, pp. 491–503. DOI: 10.1007/11431855_34.

[CA05] K. Czarnecki and M. Antkiewicz. "Mapping Features to Models: A Template Approach Based on Superimposed Variants". In: *Generative Programming and Component Engineering*. Ed. by R. Glück and M. Lowry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 422–437.

[CC06] H. Cervantes and S. Charleston-Villalobos. "Using a Lightweight Workflow Engine in a Plugin-Based Product Line Architecture". In: *Component-Based Software Engineering*. Ed. by I. Gorton, G. T. Heineman, I. Crnković, H. W. Schmidt, J. A. Stafford, C. Szyperski, and K. Wallnau. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 198–205.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007. DOI: 10.1007/978-3-540-71999-1.

[CEC00] K. Czarnecki, U. W. Eisenecker, and K. Czarnecki. *Generative programming: methods, tools, and applications*. Vol. 16. Addison Wesley Reading, 2000.

[CHE05] K. Czarnecki, S. Helsen, and U. Eisenecker. "Formalizing cardinality-based feature models and their specialization". In: *Software Process: Improvement and Practice* 10.1 (2005), pp. 7–29. DOI: 10.1002/spip.213.

[CHS11] D. Clarke, M. Helvensteijn, and I. Schaefer. "Abstract delta modeling". In: *ACM Sigplan Notices* 46.2 (2011), pp. 13–22.

[Cla10] A. Classen. *Modelling with FTS: a Collection of Illustrative Examples*. English. Tech. rep. 2010. URL: https://researchportal.unamur.be/en/publications/modelling-with-fts-a-collection-of-illustrative-examples.

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[CP06] K. Czarnecki and K. Pietroszek. "Verifying Feature-Based Model Templates against Well-Formedness OCL Constraints". In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 211–220. DOI: 10.1145/1173706.1173738.

[CRE+08] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. "Automating Co-evolution in Model-Driven Engineering". In: *2008 12th Intl. IEEE Enterprise Distributed Object Computing Conference*. Sept. 2008, pp. 222–231.

[CW07] K. Czarnecki and A. Wasowski. "Feature Diagrams and Logics: There and Back Again". In: *11th International Software Product Line Conference (SPLC 2007)*. 2007, pp. 23–34.

[DDP17a] N. Dintzner, A. van Deursen, and M. Pinzger. "FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems". In: *Empirical Software Engineering* (Nov. 2017). DOI: 10.1007/s10664-017-9557-6.

[DDP17b] N. Dintzner, A. Deursen, and M. Pinzger. "Analysing the Linux Kernel Feature Model Changes Using FMDiff". In: *Softw. Syst. Model.* 16.1 (Feb. 2017), pp. 55–76. DOI: 10.1007/s10270-015-0472-2.

[DRB+13] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. "An Exploratory Study of Cloning in Industrial Software Product Lines". In: *2013 17th European Conference on Software Maintenance and Reengineering*. 2013, pp. 25–34.

[EBL+10] C. Elsner, G. Botterweck, D. Lohmann, and W. Schröder-Preikschat. "Variability in Time - Product Line Variability and Evolution Revisited". In: *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*. Ed. by D. Benavides, D. S. Batory, and P. Grünbacher. Vol. 37. ICB-Research Report. Universität Duisburg-Essen, 2010, pp. 131–137. URL: http://www.vamos-workshop.net/proceedings/VaMoS%5C_2010%5C_Proceedings.pdf.

[EH85] E. Emerson and J. Y. Halpern. "Decision procedures and expressiveness in the temporal logic of branching time". In: *Journal of Computer and System Sciences* 30.1 (1985), pp. 1–24. DOI: https://doi.org/10.1016/0022-0000(85)90001-7.

[EHK+02] G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. "Consistency-Preserving Model Evolution through Transformations". In: *UML 2002 — The Unified Modeling Language*. Ed. by J.-M. Jézéquel, H. Hussmann, and S. Cook. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 212–227.

[EKS15] S. El-Sharkawy, A. Krafczyk, and K. Schmid. "Analysing the Kconfig Semantics and Its Analysis Tools". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 45–54. DOI: 10.1145/2814204.2814222.

[EPH09] A. O. Elfaki, S. Phon-Amnuaisuk, and C. K. Ho. "Using First Order Logic to Validate Feature Model". In: *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings*. 2009, pp. 169–172.

[EVC+07] P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, and T. D'Hondt. "Change-Oriented Software Engineering". In: *Proceedings of the Intl. Conference on Dynamic Languages*. ACM. 2007.

[FBG+13] A. Felfernig, D. Benavides, J. A. Galindo, and F. Reinfrank. "Towards Anomaly Explanation in Feature Models". In: *Proceedings of the 15th International Configuration Workshop, Vienna, Austria, August 29-30, 2013*. 2013, pp. 117–124.

[FK05] W. B. Frakes and Kyo Kang. "Software reuse research: status and future". In: *IEEE Transactions on Software Engineering* 31.7 (July 2005), pp. 529–536. DOI: 10.1109/TSE.2005.85.

[Fou19] E. Foundation. *EMF Compare Project*. 2019. URL: http://www.eclipse.org/emf/compare.

[Fow99] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. URL: http://martinfowler.com/books/refactoring.html.

[Gam95] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[Gam96] E. Gamma. "The extension objects pattern". In: *Proceedings of the 1996 Conference on Pattern Languages of Programs (PLoP 96)*. 1996.

[GBT+] J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, and A. Ruiz-Cortés. "Automated analysis of feature models: Quo vadis?" In: *Computing* (), pp. 1–47.

[GD06] T. Gîrba and S. Ducasse. "Modeling history to analyze software evolution". In: *Journal of Software Maintenance and Evolution: Research and Practice* 18.3 (2006), pp. 207–236. DOI: 10.1002/smr.325.

[GF11] N. Gamez and L. Fuentes. "Software Product Line Evolution with Cardinality-Based Feature Models". In: *Top Productivity through Software Reuse*. Ed. by K. Schmid. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 102–118.

[GJC+09] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. "Managing Model Adaptation by Precise Detection of Metamodel Changes". In: *Model Driven Architecture - Foundations and Applications*. Ed. by R. F. Paige, A. Hartman, and A. Rensink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 34–49.

[GMB06] R. Gheyi, T. Massoni, and P. Borba. "A theory for feature models in alloy". In: *First alloy workshop*. Citeseer. 2006, pp. 71–80.

[Gmb06] pure-systems GmbH. *Variant Management with pure::variants*. Tech. rep. pure-systems GmbH, 2006. URL: https://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf.

[GPF+96] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. *Algorithms for the satisfiability (SAT) problem: A survey*. Tech. rep. Cincinnati Univ oh Dept of Electrical and Computer Engineering, 1996.

[GRE10] I. Groher, A. Reder, and A. Egyed. "Incremental Consistency Checking of Dynamic Constraints". In: *Fundamental Approaches to Software Engineering*. Ed. by D. S. Rosenblum and G. Taentzer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 203–217.

[Gro16] O. M. Group. *OMG Meta Object Facility (MOF) Core Specification*. Tech. rep. Object Management Group, 2016.

[GST16] O. Guthmann, O. Strichman, and A. Trostanetski. "Minimal unsatisfiable core extraction for SMT". In: *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. 2016, pp. 57–64.

[GTA+19] K. Gomes, L. Teixeira, T. Alves, M. Ribeiro, and R. Gheyi. "Characterizing Safe and Partially Safe Evolution Scenarios in Product Lines: An Empirical Study". In: *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*. VAMOS '19. Leuven, Belgium: Association for Computing Machinery, 2019. DOI: 10.1145/3302333.3302346.

[GW10] J. Guo and Y. Wang. "Towards Consistent Evolution of Feature Models". In: *Software Product Lines: Going Beyond*. Ed. by J. Bosch and J. Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 451–455.

[GWT+12] J. Guo, Y. Wang, P. Trinidad, and D. Benavides. "Consistency maintenance for evolving feature models". In: *Expert Systems with Applications* 39.5 (2012), pp. 4987–4998. DOI: https://doi.org/10.1016/j.eswa.2011.10.014.

[HAR87] D. HAREL. "STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS". In: *Science of Computer Programming* 8 (1987), pp. 231–274.

[HBJ09] M. Herrmannsdoerfer, S. Benz, and E. Juergens. "COPE - Automating Coupled Evolution of Metamodels and Models". In: *ECOOP 2009 – Object-Oriented Programming*. Ed. by S. Drossopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 52–76.

[Hem08a] A. Hemakumar. "Finding Contradictions in Feature Models". In: *Proceedings of the 12th International Software Product Line Conference (SPLC)*. 2008, pp. 183–190.

[Hem08b] A. Hemakumar. "Finding Contradictions in Feature Models". In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*. Ed. by S. Thiel and K. Pohl. Lero Int. Science Centre, University of Limerick, Ireland, 2008, pp. 183–190.

[HJS+10] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. "Closing the Gap between Modelling and Java". In: *Software Language Engineering*. Ed. by M. van den Brand, D. Gašević, and J. Gray. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 374–383.

[HK10] M. Herrmannsdoerfer and M. Koegel. "Towards a Generic Operation Recorder for Model Evolution". In: *Proceedings of the 1st Intl. Workshop on Model Comparison in Practice*. Malaga, Spain: ACM, 2010, pp. 76–81.

[HNL+19] D. Hinterreiter, M. Nieke, L. Linsbauer, C. Seidl, H. Prähofer, and P. Grünbacher. "Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 115–128. DOI: 10.1145/3357765.3359515.

[HNS+20] A. Hoff, M. Nieke, C. Seidl, E. H. Saether, I. M. Sandberg, C. C. Din, I. C. Yu, and I. Schaefer. "Consistency-Preserving Evolution Planning on Feature Models". In: *Proceedings of the 24th International Systems and Software Product Line Conference - Volume A*. SPLC '20. Montréal, Canada: Association for Computing Machinery, 2020. DOI: https://doi.org/10.1145/3382025.3414964.

[HPL+18] D. Hinterreiter, H. Prähofer, L. Linsbauer, P. Grünbacher, F. Reisinger, and A. Egyed. "Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2018, pp. 107–114.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[HRR+11] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. "Delta Modeling for Software Architectures". In: Feb. 2011, pp. 1–10.

[Jih10] C. Jihong. "Knowledge transfer processes for different experience levels of knowledge recipients at an offshore technical support center". In: 23.1 (Jan. 2010), pp. 54–79. DOI: 10.1108/09593841011022546.

[JK07] M. Janota and J. Kiniry. "Reasoning about Feature Models in Higher-Order Logic". In: *11th International Software Product Line Conference (SPLC 2007)*. 2007, pp. 13–22.

[Joh92] D. S. Johnson. "The NP-completeness column: an ongoing guide". In: *Journal of algorithms* 13.3 (1992), pp. 502–524.

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann. "Granularity in software product lines". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008, pp. 311–320.

[KAT16] M. Kowal, S. Ananieva, and T. Thüm. "Explaining anomalies in feature models". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*. 2016, pp. 132–143. DOI: 10.1145/2993236.2993248.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[KGR+11] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. "Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 805–824. DOI: 10.1145/2048066.2048128.

[KGS18] C. Kröher, L. Gerling, and K. Schmid. "Identifying the Intensity of Variability Changes in Software Product Line Evolution". In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. SPLC '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 54–64. DOI: 10.1145/3233027.3233032.

[KHH+09] M. Koegel, M. Herrmannsdoerfer, J. Helming, and Y. Li. "State-based vs. operation-based change tracking". In: *Proceedings of MODELS*. Vol. 9. 2009.

[KKK+19] E. Kuiter, S. Krieter, J. Krüger, T. Leich, and G. Saake. "Foundations of Collaborative, Real-Time Feature Modeling". In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. SPLC '19. Paris, France: Association for Computing Machinery, 2019, pp. 257–264. DOI: 10.1145/3336294.3336308.

[KKO+12] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. "Understanding model evolution through semantically lifting model differences with SiLift". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Sept. 2012, pp. 638–641. DOI: 10.1109/ICSM.2012.6405342.

[KKT13] T. Kehrer, U. Kelter, and G. Taentzer. "Consistency-preserving edit scripts in model versioning". In: *2013 28th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE)*. Nov. 2013, pp. 191–201.

[KMS+83] J. Kramer, J. Magee, M. Sloman, and A. Lister. "CONIC: an integrated approach to distributed computer control systems". In: *IEE Proceedings E - Computers and Digital Techniques* 130.1 (1983), pp. 1–.

[KSR13] D. Kramer, C. S. Sauer, and T. Roth-Berghofer. "Towards Explanation Generation using Feature Models in Software Product Lines". In: *Proceedings of 9th Workshop on Knowledge Engineering and Software Engineering (KESE9) co-located with the 36th German Conference on Artificial Intelligence (KI2013), Koblenz, Germany, September 17, 2013*. 2013.

[KSW16] W. Kessentini, H. Sahraoui, and M. Wimmer. "Automated Metamodel/Model Co-evolution Using a Multi-objective Optimization Approach". In: *Proceedings of the 12th European Conference on Modelling Foundations and Applications*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 138–155.

[KTM+17] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer. "Is There a Mismatch between Real-World Feature Models and Product-Line Research?" In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 291–302. DOI: 10.1145/3106237.3106252.

[KTS+18] S. Krieter, T. Thüm, S. Schulze, R. Schröter, and G. Saake. "Propagating Configuration Decisions with Modal Implication Graphs". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. ACM, New York, NY, USA, 2018. DOI: 10.1145/3180155.3180159.

[LAL+10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. "An analysis of the variability in forty preprocessor-based software product lines". In: *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*. Vol. 1. IEEE. 2010, pp. 105–114.

[LG96] N. Leveson and S. Goetsch. "Safeware: System Safety and Computers". In: *Medical Physics-New York-Institute of Physics* 23.10 (1996), p. 1821.

[Liv11] S. Livengood. "Issues in Software Product Line Evolution: Complex Changes in Variability Models". In: *Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering*. PLEASE '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 6–9. DOI: 10.1145/1985484.1985487.

[LKS16] S. Lity, M. Kowal, and I. Schaefer. "Higher-Order Delta Modeling for Software Product Line Evolution". In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. FOSD 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 39–48. DOI: 10.1145/3001867.3001872.

[LLL+13] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer. *Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study*. TU Braunschweig, Institut für Softwaretechnik und Fahrzeuginformatik, 2013. URL: https://www.isf.cs.tu-bs.de/cms/team/lity/bcs_tubs_tech_rep_V1_4.pdf.

[LS08] M. H. Liffiton and K. A. Sakallah. "Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints". In: *J. Autom. Reasoning* 40.1 (2008), pp. 1–33.

[LSW15] U. Lesta, I. Schaefer, and T. Winkelmann. "Detecting and Explaining Conflicts in Attributed Feature Models". In: *Proceedings 6th Workshop on Formal Methods and Analysis in SPL Engineering, FMSPLE@ETAPS 2015, London, UK, 11 April 2015*. 2015, pp. 31–43.

[MB08] L. de Moura and N. Bjrner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and J. Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[ME08] R. Mitschke and M. Eichberg. "Supporting the evolution of software product lines". In: *ECMDA Traceability Workshop (ECMDA-TW)*. Citeseer. 2008, pp. 87–96.

[Mes00] J. Meseguer. "Rewriting Logic and Maude: Concepts and Applications". In: *Rewriting Techniques and Applications*. Ed. by L. Bachmair. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–26.

[Mes12] J. Meseguer. "Twenty years of rewriting logic". In: *The Journal of Logic and Algebraic Programming* 81.7-8 (2012), pp. 721–781. DOI: 10.1016/j.jlap.2012.06.003.

[ML04] T. von der Maßen and H. Lichter. "Deficiencies in feature models". In: *Workshop on Software Variability Management for Product Derivation-Towards Tool Support*. Vol. 44. 2004.

[MM02] N. Martí-Oliet and J. Meseguer. "Rewriting Logic as a Logical and Semantic Framework". In: *Handbook of Philosophical Logic*. Ed. by D. M. Gabbay and F. Guenthner. Dordrecht: Springer Netherlands, 2002, pp. 1–87. DOI: 10.1007/978-94-017-0464-9_1.

[MNM+18] M. Mukelabai, D. Nešić, S. Maro, T. Berger, and J.-P. Steghöfer. "Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: ACM, 2018, pp. 155–166. DOI: 10.1145/3238147.3238201.

[MNS+17] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. "Anomaly Detection and Explanation in Context-Aware Software Product Lines". In: *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*. SPLC '17. Sevilla, Spain: ACM, 2017, pp. 18–21. DOI: 10.1145/3109729.3109752.

[MNS+18] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. "Context-aware reconfiguration in evolving software product lines". In: *Science of Computer Programming* 163 (2018), pp. 139–159. DOI: https://doi.org/10.1016/j.scico.2018.05.002.

[MR14a] V. Montaghami and D. Rayside. "Staged Evaluation of Partial Instances in a Relational Model Finder". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Y. Ait Ameur and K.-D. Schewe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 318–323.

[MR14b] V. Montaghami and D. Rayside. "Staged evaluation of partial instances in a relational model finder". In: *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer. 2014, pp. 318–323.

[MSR+19] M. Marques, J. Simmonds, P. O. Rossel, and M. C. Bastarrica. "Software product line evolution: A systematic literature review". In: *Information and Software Technology* 105 (2019), pp. 190–208. DOI: https://doi.org/10.1016/j.infsof.2018.08.014.

[MTS+17] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. DOI: 10.1007/978-3-319-61443-4.

[MWC09] M. Mendonca, A. Wąsowski, and K. Czarnecki. "SAT-Based Analysis of Feature Models is Easy". In: *Proceedings of the 13th International Software Product Line Conference*. SPLC '09. San Francisco, California, USA: Carnegie Mellon University, 2009, pp. 231–240.

[NBA+15] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. "Safe evolution templates for software product lines". In: *Journal of Systems and Software* 106 (2015), pp. 42–58. DOI: 10.1016/j.jss.2015.04.024.

[NDT+13] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann. "Linux Variability Anomalies: What Causes Them and How Do They Get Fixed?" In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 111–120. DOI: 10.5555/2487085.2487112.

[NES17] M. Nieke, G. Engel, and C. Seidl. "DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-aware Software Product Lines". In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '17. Eindhoven, Netherlands: ACM, 2017, pp. 92–99. DOI: 10.1145/3023956.3023962.

[NHS+20] M. Nieke, A. Hoff, C. Seidl, and I. Schaefer. "Automated Metamodel Augmentation for Seamless Model Evolution Tracking and Planning". In: *Journal of Computer Languages* (2020). In minor revision.

[NHS19] M. Nieke, A. Hoff, and C. Seidl. "Automated Metamodel Augmentation for Seamless Model Evolution Tracking and Planning". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 68–80. DOI: 10.1145/3357765.3359526.

[NLS18] S. Nahrendorf, S. Lity, and I. Schaefer. *Applying Higher-Order Delta Modeling for the Evolution of Delta-Oriented Software Product Lines*. TU Braunschweig, Institut für Softwaretechnik und Fahrzeuginformatik, 2018. URL: https://www.isf.cs.tu-bs.de/cms/team/lity/TUBS_Report_2018-01_Nahrendorf_et_al.pdf.

[NMS+18] M. Nieke, J. Mauro, C. Seidl, T. Thüm, I. C. Yu, and F. Franzke. "Anomaly Analyses for Feature-model Evolution". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: ACM, 2018, pp. 188–201. DOI: 10.1145/3278122.3278123.

[NNP+10] T. T. Nguyen, H. A. Nguyen, N. H. Pham, and T. N. Nguyen. "Operation-Based, Fine-Grained Version Control Model for Tree-Based Representation". In: *Fundamental Approaches to Software Engineering*. Ed. by D. S. Rosenblum and G. Taentzer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 74–90.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T)". In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 937–977.

[NSS16] M. Nieke, C. Seidl, and S. Schuster. "Guaranteeing Configuration Validity in Evolving Software Product Lines". In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 73–80. DOI: 10.1145/2866614.2866625.

[NST+20a] M. Nieke, G. Sampaio, T. Thüm, C. Seidl, L. Teixeira, and I. Schaefer. "Guided Configuration Evolution for Product Lines". In: *Software and Systems Modeling* (2020). Submitted.

[NST+20b] M. Nieke, G. Sampaio, T. Thüm, C. Seidl, L. Teixeira, and I. Schaefer. "GuyDance: Guiding the Evolution of Product-Line Configurations". In: *Proceedings of the 24th International Systems and Software Product Line Conference - Volume B*. SPLC '20. Accepted for publication. 2020. DOI: https://doi.org/10.1145/3382026.3425769.

[NST18] M. Nieke, C. Seidl, and T. Thüm. "Back to the Future: Avoiding Paradoxes in Feature-model Evolution". In: *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 2*. SPLC '18. Gothenburg, Sweden: ACM, 2018, pp. 48–51. DOI: 10.1145/3236405.3237201.

[NTS+11] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza, and P. Borba. "Investigating the Safe Evolution of Software Product Lines". In: *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. GPCE '11. Portland, Oregon, USA: ACM, 2011, pp. 33–42. DOI: 10.1145/2047862.2047869.

[OGB19] J. Oh, P. Gazzillo, and D. Batory. "T-wise Coverage by Uniform Sampling". In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. SPLC '19. Paris, France: ACM, 2019, pp. 84–87. DOI: 10.1145/3336294.3342359.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. "PVS: A prototype verification system". In: *International Conference on Automated Deduction*. Springer. 1992, pp. 748–752.

[PBD+12] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. "Model-driven support for product line evolution on feature level". In: *Journal of Systems and Software* 85.10 (2012). Automated Software Evolution, pp. 2261–2274. DOI: https://doi.org/10.1016/j.jss.2011.08.008.

[PBL05] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Berlin Heidelberg, Aug. 3, 2005. 496 pp.

[PCA+13] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. "Feature-oriented Software Evolution". In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '13. Pisa, Italy: ACM, 2013, 17:1–17:8.

[PDŠ12] P. Paskevicius, R. Damasevicius, and V. Štuikys. "Change Impact Analysis of Feature Models". In: *Information and Software Technologies*. Ed. by T. Skersys, R. Butleris, and R. Butkiene. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 108–122.

[Plo04] G. D. Plotkin. "The origins of structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60 (2004), pp. 3–15.

[PMK+16] J. A. Pereira, P. Matuszyk, S. Krieter, M. Spiliopoulou, and G. Saake. "A Feature-based Personalized Recommender System for Product-line Configuration". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 120–131. DOI: 10.1145/2993236.2993249.

[PSF+18] J. A. Pereira, S. Schulze, E. Figueiredo, and G. Saake. "N-Dimensional Tensor Factorization for Self-Configuration of Software Product Lines at Runtime". In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. SPLC '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 87–97. DOI: 10.1145/3233027.3233039.

[PSK+18] J. A. Pereira, S. Schulze, S. Krieter, M. Ribeiro, and G. Saake. "A Context-Aware Recommender System for Extended Software Product Line Configurations". In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. VAMOS 2018. Madrid, Spain: Association for Computing Machinery, 2018, pp. 97–104. DOI: 10.1145/3168365.3168373.

[PTD+16] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. "Coevolution of variability models and related software artifacts". In: *Empirical Software Engineering* 21.4 (2016), pp. 1744–1793. DOI: 10.1007/s10664-015-9364-x.

[PTR+19] T. Pett, T. Thüm, T. Runge, S. Krieter, M. Lochau, and I. Schaefer. "Product Sampling for Product Lines: The Scalability Challenge". In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. SPLC '19. Paris, France: ACM, 2019, pp. 78–83. DOI: 10.1145/3336294.3336322.

[QPB+14] C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, and G. Botterweck. "Consistency Checking for the Evolution of Cardinality-Based Feature Models". In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. SPLC '14. Florence, Italy: Association for Computing Machinery, 2014, pp. 122–131. DOI: 10.1145/2648511.2648524.

[RGM+14] L. Rincón, G. L. Giraldo, R. Mazo, and C. Salinesi. "An ontological rule-based approach for analyzing dead and false optional features in feature models". In: *Electronic notes in theoretical computer science* 302 (2014), pp. 111–132.

[Ric53] H. G. Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[RJW+09] T. Reinhard, S. Julio, S.-P. Wolfgang, and L. Daniel. "Dead or Alive: Finding Zombie Features in the Linux Kernel". In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. FOSD '09. New York, NY, USA: ACM, 2009, pp. 81–86. DOI: 10.1145/1629716.1629732.

[RKP+10] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. "Model Migration with Epsilon Flock". In: *Theory and Practice of Model Transformations*. Ed. by L. Tratt and M. Gogolla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 184–198.

[SB99] M. Svahnberg and J. Bosch. "Evolution in Software Product Lines: Two Cases". In: *Journal of Software Maintenance* 11.6 (Nov. 1999), pp. 391–422. DOI: 10.1002/(SICI)1096-908X(199911/12)11:6<391::AID-SMR199>3.0.CO;2-8.

[SBB+10] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. "Delta-oriented programming of software product lines". In: *Software Product Lines: Going Beyond* (2010), pp. 77–91.

[SBT16] G. Sampaio, P. Borba, and L. Teixeira. "Partially Safe Evolution of Software Product Lines". In: *Proceedings of the 20th International Systems and Software Product Line Conference*. SPLC '16. Beijing, China: ACM, 2016, pp. 124–133. DOI: 10.1145/2934466.2934482.

[SBT19] G. Sampaio, P. Borba, and L. Teixeira. "Partially safe evolution of software product lines". In: *Journal of Systems and Software* 155 (2019), pp. 17–42. DOI: https://doi.org/10.1016/j.jss.2019.04.051.

[SD10] I. Schaefer and F. Damiani. "Pure Delta-Oriented Programming". In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. FOSD '10. Eindhoven, The Netherlands: Association for Computing Machinery, 2010, pp. 49–56. DOI: 10.1145/1868688.1868696.

[SHA12] C. Seidl, F. Heidenreich, and U. Aundefinedmann. "Co-Evolution of Models and Feature Mapping in Software Product Lines". In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC '12. Salvador, Brazil: Association for Computing Machinery, 2012, pp. 76–85. DOI: 10.1145/2362536.2362550.

[SHT+07] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. "Generic semantics of feature diagrams". In: *Computer Networks* 51.2 (2007), pp. 456–479.

[SHT06] P. Schobbens, P. Heymans, and J. Trigaux. "Feature Diagrams: A Survey and a Formal Semantics". In: *14th IEEE International Requirements Engineering Conference (RE'06)*. 2006, pp. 139–148.

[SLB+10] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. "The Variability Model of The Linux Kernel". In: *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*. Ed. by D. Benavides, D. S. Batory, and P. Grünbacher. Vol. 37. ICB-Research Report. Universität Duisburg-Essen, 2010, pp. 45–51. URL: http://www.vamos-workshop.net/proceedings/VaMoS%5C_2010%5C_Proceedings.pdf.

[SPB+12] M. Schubanz, A. Pleuss, G. Botterweck, and C. Lewerentz. "Modeling Rationale over Time to Support Product Line Evolution Planning". In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '12. Leipzig, Germany: ACM, 2012, pp. 193–199. DOI: 10.1145/2110147.2110169.

[SPP+13] M. Schubanz, A. Pleuss, L. Pradhan, G. Botterweck, and A. K. Thurimella. "Model-Driven Planning and Monitoring of Long-Term Software Product Line Evolution". In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '13. Pisa, Italy: Association for Computing Machinery, 2013. DOI: 10.1145/2430502.2430527.

[SRC+12] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. "Software diversity: state of the art and perspectives". In: *International Journal on Software Tools for Technology Transfer* 14.5 (Oct. 2012), pp. 477–495. DOI: 10.1007/s10009-012-0253-y.

[SRS13] S. Schulze, O. Richers, and I. Schaefer. "Refactoring Delta-oriented Software Product Lines". In: *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*. AOSD '13. Fukuoka, Japan: ACM, 2013, pp. 73–84. DOI: 10.1145/2451436.2451446.

[SSA14a] C. Seidl, I. Schaefer, and U. Aßmann. "DeltaEcore - A Model-Based Delta Language Generation Framework". In: *Modellierung 2014, 19.-21. März 2014, Wien, Österreich*. 2014, pp. 81–96. URL: http://subs.emis.de/LNI/Proceedings/Proceedings225/article2.html.

[SSA14b] C. Seidl, I. Schaefer, and U. Aundefinedmann. "Capturing Variability in Space and Time with Hyper Feature Models". In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: Association for Computing Machinery, 2014. DOI: 10.1145/2556624.2556625.

[SSA14c] C. Seidl, I. Schaefer, and U. Aundefinedmann. "Integrated Management of Variability in Space and Time in Software Families". In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. SPLC '14. Florence, Italy: Association for Computing Machinery, 2014, pp. 22–31. DOI: 10.1145/2648511.2648514.

[SSK+20] J. Sprey, C. Sundermann, S. Krieter, M. Nieke, J. Mauro, T. Thüm, and I. Schaefer. "SMT-Based Variability Analyses in FeatureIDE". In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS '20. Magdeburg, Germany: Association for Computing Machinery, 2020. DOI: 10.1145/3377024.3377036.

[STK+12] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. "Variant-preserving Refactoring in Feature-oriented Software Product Lines". In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '12. Leipzig, Germany: ACM, 2012, pp. 73–81. DOI: 10.1145/2110147.2110156.

[SW16] F. Schwäger and B. Westfechtel. "SuperMod: Tool support for collaborative filtered model-driven software product line engineering". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2016, pp. 822–827.

[TAK+14] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. "A Classification and Survey of Analysis Strategies for Software Product Lines". In: *ACM Comput. Surv.* 47.1 (June 2014), 6:1–6:45. DOI: 10.1145/2580950.

[TBK09] T. Thum, D. Batory, and C. Kastner. "Reasoning about edits to feature models". In: *2009 IEEE 31st International Conference on Software Engineering*. May 2009, pp. 254–264.

[TBR+06] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and M. Toro. "Explanations for agile feature models". In: *Procceedings of the 1st International Workshop on Agile Product Line Engineering (APLE'06)*. 2006, p. 44.

[TEL+14] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. "A fundamental approach to model versioning based on graph modifications: from theory to implementation". In: *Software & Systems Modeling* 13.1 (Feb. 2014), pp. 239–272.

[TKE+11] T. Thüm, C. Kastner, S. Erdweg, and N. Siegmund. "Abstract Features in Feature Modeling". In: *2011 15th International Software Product Line Conference*. 2011, pp. 191–200.

[TKS18a] T. Thüm, S. Krieter, and I. Schaefer. "Product Configuration in the Wild: Strategies for Conflicting Decisions in Web Configurators". In: *Proceedings of the 20th Configuration Workshop, Graz, Austria, September 27-28, 2018*. 2018, pp. 1–8.

[TKS18b] T. Thüm, S. Krieter, and I. Schaefer. "Product Configuration in the Wild: Strategies for Conflicting Decisions in Web Configurators". In: *Proceedings of the 20th Configuration Workshop, Graz, Austria, September 27-28, 2018*. Ed. by A. Felfernig, J. Tiihonen, L. Hotz, and M. Stettinger. Vol. 2220. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 1–8. URL: http://ceur-ws.org/Vol-2220/01%5C_CONFWS18%5C_paper%5C_35.pdf.

[TLD+11] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. "Configuration Coverage in the Analysis of Large-scale System Software". In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. PLOS '11. Cascais, Portugal: ACM, 2011, 2:1–2:5. DOI: 10.1145/2039239.2039242.

[TLS+11] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. "Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem". In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 47–60. DOI: 10.1145/1966445.1966451.

[Tri12] P. Trinidad Martin-Arroyo. "Automating the analysis of stateful feature models". PhD thesis. University of Seville, 2012.

[VGH+12] M. Vierhauser, P. Grünbacher, W. Heider, G. Holl, and D. Lettner. "Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines". In: *Model Driven Engineering Languages and Systems*. Ed. by R. B. France, J. Kazmeier, R. Breu, and C. Atkinson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 531–545.

[VSB+13] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[VV98] J. Vlissides and J. M. Vlissides. *Pattern hatching: design patterns applied*. Addison-Wesley Reading, 1998.

[Wac07] G. Wachsmuth. "Metamodel Adaptation and Model Co-adaptation". In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by E. Ernst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 600–624.

[Wel82] W. J. Welch. "Algorithmic complexity: three NP-hard problems in computational statistics". In: *Journal of Statistical Computation and Simulation* 15.1 (1982), pp. 17–25.

[WGS+14] J. White, J. A. Galindo, T. Saxena, B. Dougherty, D. Benavides, and D. C. Schmidt. "Evolving feature model configurations in software product lines". In: *Journal of Systems and Software* 87 (2014), pp. 119–136. DOI: https://doi.org/10.1016/j.jss.2013.10.010.

[WLS+16] M. Weckesser, M. Lochau, T. Schnabel, B. Richerzhagen, and A. Schürr. "Mind the Gap! Automated Anomaly Detection for Potentially Unbounded Cardinality-Based Feature Models". In: *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering - Volume 9633*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 158–175. DOI: 10.1007/978-3-662-49665-7_10.

[WPX+13] B. Wang, L. Passos, Y. Xiong, K. Czarnecki, H. Zhao, and W. Zhang. "SmartFixer: Fixing Software Configurations Based on Dynamic Priorities". In: *Proceedings of the 17th International Software Product Line Conference*. SPLC '13. Tokyo, Japan: ACM, 2013, pp. 82–90. DOI: 10.1145/2491627.2491640.

[WRS+17] D. Wille, T. Runge, C. Seidl, and S. Schulze. "Extractive Software Product Line Engineering Using Model-based Delta Module Generation". In: *Proceedings of the Eleventh Intl. Workshop on Variability Modelling of Software-intensive Systems*. Eindhoven, Netherlands: ACM, 2017, pp. 36–43.

[WSB+08] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. "Automated Diagnosis of Product-Line Configuration Errors in Feature Models". In: *2008 12th International Software Product Line Conference*. Sept. 2008, pp. 225–234. DOI: 10.1109/SPLC.2008.16.

[XHS+12] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. "Generating Range Fixes for Software Configuration". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 58–68. DOI: 10.1109/ICSE.2012.6227206.

[XS06] Z. Xing and E. Stroulia. "Refactoring Detection based on UMLDiff Change-Facts Queries". In: *13th Working Conference on Reverse Engineering*. Oct. 2006, pp. 263–274.

[XXJ10] Y. Xue, Z. Xing, and S. Jarzabek. "Understanding Feature Evolution in a Family of Product Variants". In: *2010 17th Working Conference on Reverse Engineering*. 2010, pp. 109–118.

[XZH+13] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. "Do Not Blame Users for Misconfigurations". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 244–259. DOI: 10.1145/2517349.2522727.

[ZE14] S. Zhang and M. D. Ernst. "Which Configuration Option Should I Change?" In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 152–163. DOI: 10.1145/2568225.2568251.

[ZRL16] A. Ziegler, V. Rothberg, and D. Lohmann. "Analyzing the Impact of Feature Changes in Linux". In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 25–32. DOI: 10.1145/2866614.2866618.

Curriculum Vitae

Michael Nieke

* 10.06.1989 in Northeim

Academic Employment

05/2019 - 07/2019	**Parental Leave**
Since 06/2015	**Doctoral Student** Technische Universität Braunschweig, Germany
09/2014 - 04/2015	**Internship to write my Master's thesis** R&D Department of Volkswagen AG, Germany
02/2013 - 09/2014	**Student Assistant** Technische Universität Braunschweig, Germany Responsibilities: Support for the Lecture Software Architecture, Software Development for Research Projects and two Industry Projects
06/2012 - 10/2012	**Internship to write my Bachelor's thesis** R&D Department of Volkswagen AG, Germany

Education

Since 06/2015	**Dissertation (Dr.-Ing.)** Technische Universität Braunschweig, Germany Title: Modeling Consistent Software Product Line Evolution
04/2013 - 05/2015	**Master of Science (M.Sc.) in Computer Science** Technische Universität Braunschweig, Germany Title: Proof-Carrying-Apps – A Case Study in the Infotainment Domain Grade: 1.1 (with honors)
10/2009 - 03/2013	**Bachelor of Science (B.Sc.) in Computer Science** Technische Universität Braunschweig, Germany Title: Multimodal Speech– and Gesturebased Vehicle Control Grade: 2.1
08/2000 - 06/2008	**Abitur** Max-Planck-Gymnasium in Göttingen, Germany Grade: 2.3 Advanced courses: Physics, Mathematics, Computer Science
08/1998 - 07/2000	**Orientation Level** Thomas-Mann-Schule in Northeim, Germany
08/1994 - 07/1998	**Elementary School** in Sudheim, Germany

Technische Universität Carolo-Wilhelmina Braunschweig
Institut für Softwaretechnik und Fahrzeuginformatik

Mühlenpfordtstr. 23
D-38106 Braunschweig